MATLAB® 7
Graphics

MATLAB®

The MathWorks™

Accelerating the pace of engineering and science

How to Contact The MathWorks

www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support
suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.com	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
info@mathworks.com	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB® Graphics

Revision History

March 2006	Online only	New for MATLAB 7.2 (Release 2006a)
September 2006	Online only	Revised for MATLAB 7.3 (Release 2006b)
March 2007	Online only	Revised for MATLAB 7.4 (Release 2007a)
September 2007	Online only	Revised for MATLAB 7.5 (Release 2007b)
March 2008	Online only	Revised for MATLAB 7.6 (Release 2008a) This publication was previously part of the Using MATLAB® Graphics User Guide.
October 2008	Online only	Revised for MATLAB 7.7 (Release 2008b)
March 2009	Online only	Revised for MATLAB 7.8 (Release 2009a)

Contents

1 Plots and Plotting Tools

2 Data Exploration Tools

3 Annotating Graphs

4 Basic Plotting Commands

5 Creating Specialized Plots

6 Displaying Bit-Mapped Images

7 Printing and Exporting

8 Handle Graphics Objects

9 Figure Properties

10 Axes Properties

Index

Plots and Plotting Tools

If you are viewing this document in the MATLAB® Help browser, watch the Interactive Plot Creation with the Plot Tools video demo for an overview of the major functionality. It covers much of the material presented in the following sections:

Figures, Plots, and Graphs

In this section...

What Is a MATLAB Graph?

The MATLAB environment offers a variety of data plotting functions plus a set of GUI tools to create, and modify graphic displays. The GUI tools afford most of the control over graphic properties and options that typed commands such as `annotate`, `get`, and `set` provide.

A *figure* is a MATLAB window that contains graphic displays (usually data plots) and UI components. You create figures explicitly with the `figure` function, and implicitly whenever you plot graphics and no figure is active. By default, figure windows are resizable and include pull-down menus and toolbars.

A *plot* is any graphic display you can create within a figure window. Plots can display tabular data, geometric objects, surface and image objects, and annotations such as titles, legends, and colorbars. Figures can contain any number of plots. Each plot is created within a 2-D or a 3-D data space called an *axes*. You can explicitly create axes with the `axes` or `subplot` functions.

A *graph* is a plot of *data* within a 2-D or 3-D axes. Most plots made with MATLAB functions and GUIs are therefore graphs. When you graph a one-dimensional variable (e.g., `rand(100,1)`), the indices of the data vector (in this case `1:100`) become assigned as x values, and plots the data vector as y values. Some types of graphs can display more than one variable at a time, others cannot.

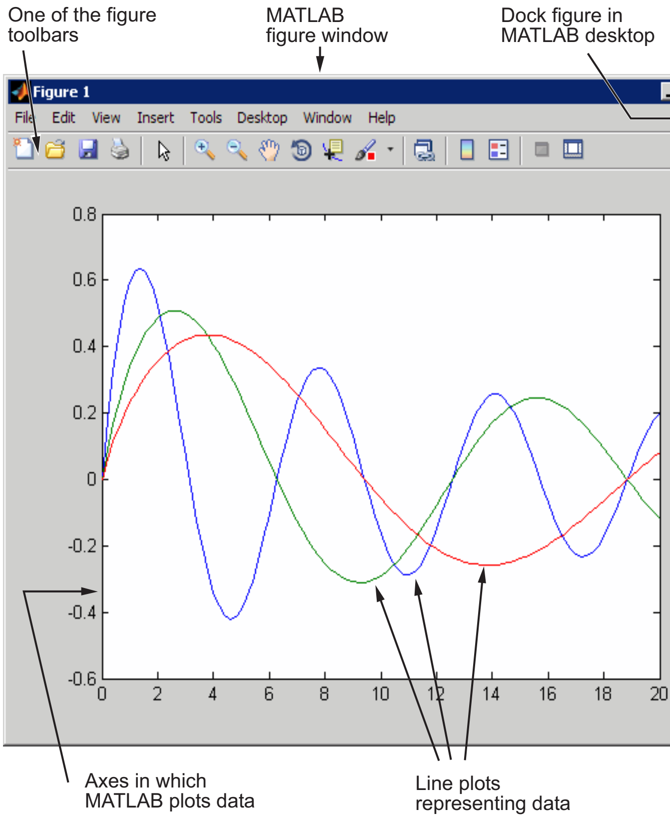
The contents and varieties of figures, plots and graphs that MATLAB can make are explained in the following sections.

Anatomy of a Graph

MATLAB plotting functions and tools direct their output to a figure window. Each figure is a separate window that you can dock in the desktop, and collect together with other plots in a *Figure Group*. To illustrate the basic components of a graph, execute the following M-code to create a plot of a family of sine curves:

```
x = [0:.2:20];
y = sin(x)./sqrt(x+1);
y(2,:) = sin(x/2)./sqrt(x+1);
y(3,:) = sin(x/3)./sqrt(x+1);
plot(x,y)
```

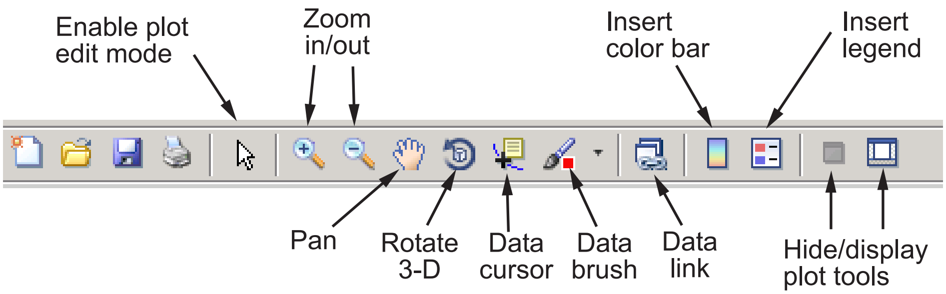
The resulting figure contains a 2-D set of axes. This graphic identifies the components and tools of a figure window.



The `plot` function uses a default line style and color to distinguish the data sets plotted in the graph. You can change the appearance of these graphic components or add annotations to the graph to present your data in a particular way.

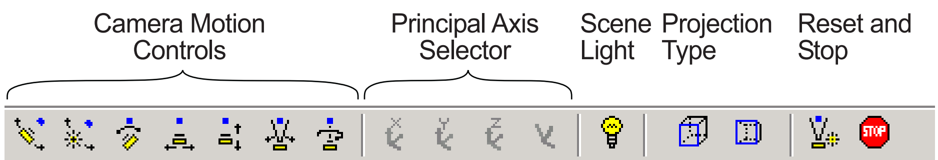
Figure Toolbars

Figure toolbars provide shortcuts to access commonly used features. These include operations such as saving and printing, plus tools for interactive zooming, panning, rotating, querying, and editing plots. The following picture shows the features available from this toolbar.

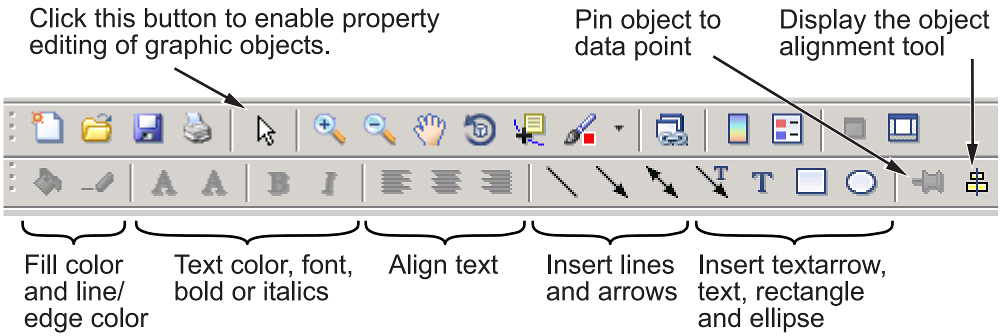


You can enable two other toolbars from the **View** menu:

- Camera Toolbar — Use for manipulating 3-D views. See "View Control with the Camera Toolbar" in the MATLAB 3-D Visualization documentation for more information.



- Plot Edit Toolbar — Use for annotation and setting object properties. See "Annotation Tools on the Plot Edit Toolbar" on page 3-3 for more information.

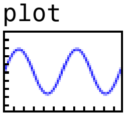
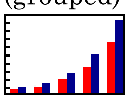
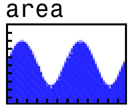
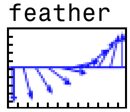
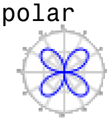
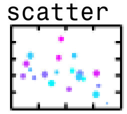


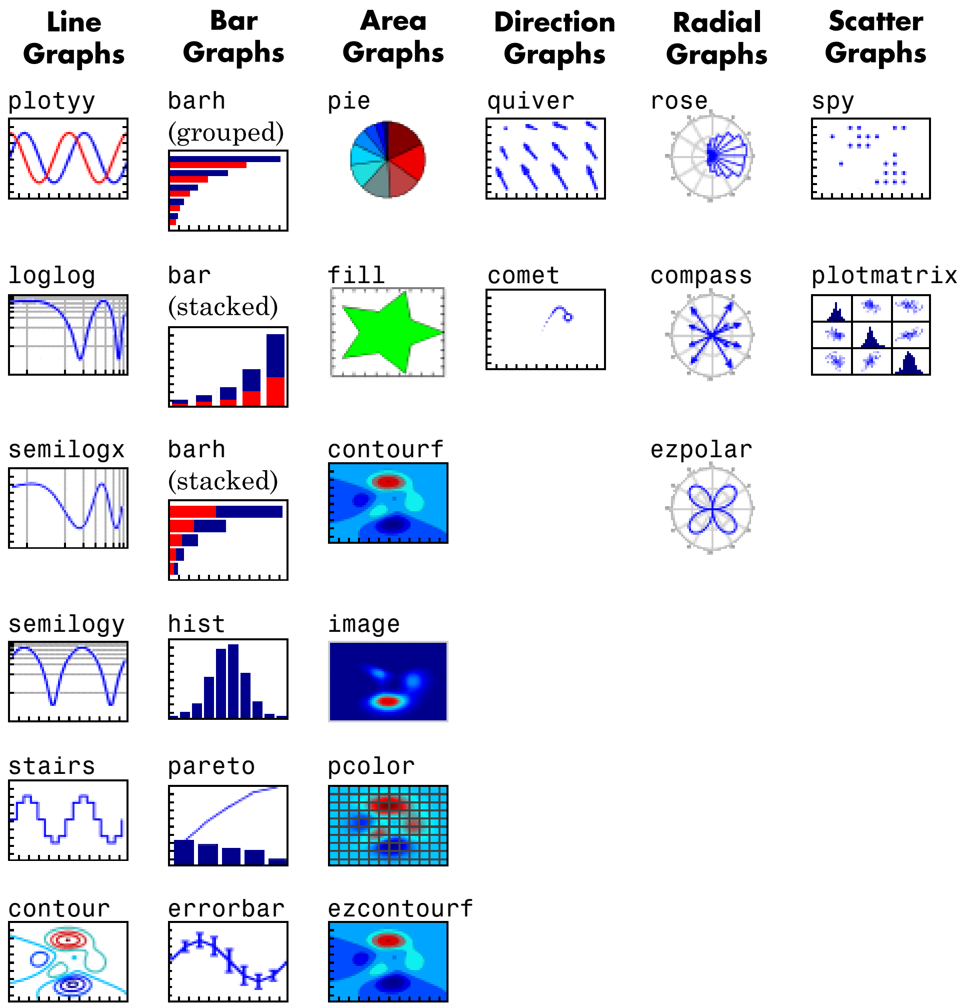
Types of MATLAB Plots

You can construct a wide variety of 2-D and 3-D MATLAB plots with very little, if any, programming required on your part. The following two tables classify and illustrate most of the kinds of plots you can create. They include line, bar, area, direction and vector field, radial, and scatter graphs. They also include 2-D and 3-D functions that generate and plot geometric shapes and objects. Most 2-D plots have 3-D analogs, and there are a variety of volumetric displays for 3-D solids and vector fields. Plot types that begin with “ez” (such as `ezsurf`) are convenience functions that can plot arguments given as functions.

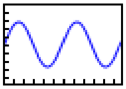
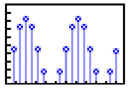
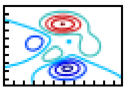
Two-Dimensional Plotting Functions

The table below shows all available MATLAB 2-D plot functions. If you are reading this online, you can click any icon to see the documentation for that function. Techniques for using many of the functions are also discussed in later sections of this document.

Line Graphs	Bar Graphs	Area Graphs	Direction Graphs	Radial Graphs	Scatter Graphs
`plot` 	`bar` (grouped) 	`area` 	`feather` 	`polar` 	`scatter` 

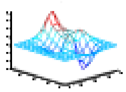
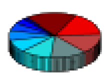
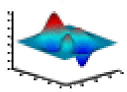
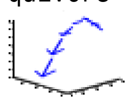
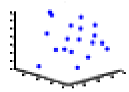
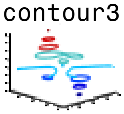
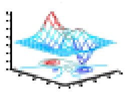
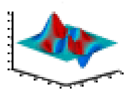
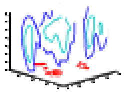
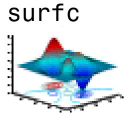
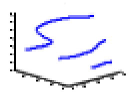


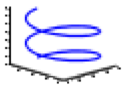
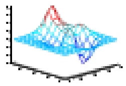
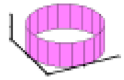
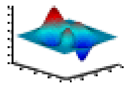
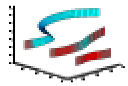
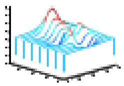
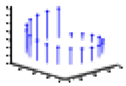
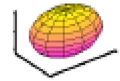
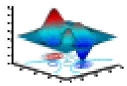
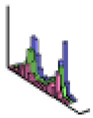
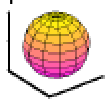
Line Graphs
Bar Graphs
Area Graphs
Direction Graphs
Radial Graphs
Scatter Graphs
plotyy
barh (grouped)
pie
quiver
rose
spy
loglog
bar (stacked)
fill
comet
compass
plotmatrix
semilogx
barh (stacked)
contourf
ezpolar
semilogy
hist
image
stairs
pareto
pcolor
contour
errorbar
ezcontourf

Line Graphs	Bar Graphs	Area Graphs	Direction Graphs	Radial Graphs	Scatter Graphs
ezplot 	stem 				
ezcontour 					

Three-Dimensional Plotting Functions

The table below shows all available MATLAB 3-D and volumetric plot functions. It includes functions that generate 3-D data (`cylinder`, `ellipsoid`, `sphere`), but most plot either arrays of data or functions. If you are reading this online, you can click any picture in the table to see the documentation for that function. For information about and examples of using 3-D plotting functions, see "Creating 3-D Graphs" in the 3-D Visualization documentation.

Line Graphs	Mesh Graphs and Bar Graphs	Area Graphs and Constructive Objects	Surface Graphs	Direction Graphs	Volumetric Graphs
plot3 	mesh 	pie3 	surf 	quiver3 	scatter3 
contour3 	meshc 	fill3 	surfl 	comet3 	coneplot 
contourslice 	meshz 	patch 	surfc 	streamslice 	streamline 

Line Graphs	Mesh Graphs and Bar Graphs	Area Graphs and Constructive Objects	Surface Graphs	Direction Graphs	Volumetric Graphs
`ezplot3` 	`ezmesh` 	`cylinder` 	`ezsurf` 		`streamribbon` 
`waterfall` 	`stem3` 	`ellipsoid` 	`ezsurfc` 		`streamtube` 
	`bar3` 	`sphere` 			
	`bar3h` 				

Choosing a Plot Type with the Plot Catalog

Most of the plotting functions shown in the previous tables are accessible through the **Figure Palette**, one of the Plot Tools you can access via the figure window **View** menu. When the Figure Palette is active and you select one, two or more variables listed within it, you can generate a plot of any appropriate type by right-clicking and selecting a plot type from the context menu that appears. The lowest item on that menu is **More Plots**. When you select **More Plots**, the *Plot Catalog* opens for you to browse through all plot types and generate one of them, either to display the variables you selected in the Figure Palette or a MATLAB expression you can specify in the Plot Catalog window. For more information, see "The Figure Palette" on page 1-18.

Plotting Tools — Interactive Plotting

In this section...

What Are Plotting Tools?

The modular, interactive plotting environment called *plotting tools* enables you to

- Create various type of graphs
- Select variables to plot directly from a workspace browser
- Easily create and manipulate subplots in the figure
- Add annotations such as arrows, lines, and text
- Set properties on graphics objects

You can open and configure plotting tools in many ways. To create a figure with the plotting tools attached, use the `plottools` command. You can also start the plotting tools from the figure toolbar by clicking the **Show Plot Tools** icon .

Remove the plotting tools from a figure using the Hide Plot Tools icon .

You can display the three basic plotting tools from the **View** menu by selecting **Figure Palette**, **Plot Browser**, or **Property Editor** .

The next section describes the individual components making up the plotting tools.

Plotting Tools Interface Overview

The Plotting Tools interface includes three panels that are associated with a figure.

- Figure Palette — Use to create and arrange subplot axes, view and plot workspace variables, and add annotations. Display the Figure Palette using the `figurepalette` command.
- Plot Browser — Use to select and control the visibility of the axes or graphic objects plotted in the figure. You can also add data to any selected axes by clicking the **Add Data** button. Display the Plot Browser using the `plotbrowser` command.
- Property Editor — Use to set common properties of the selected object. You can also open the Property Editor using the `propertyeditor` command. In the Property Editor you can click the **More Properties** button to display the Property Inspector, a GUI that displays most object properties and allows you to change any property's value (unless it is read-only). See "Accessing Object Properties with the Property Inspector" on page 1-29 for details.

Activating Plotting Tools

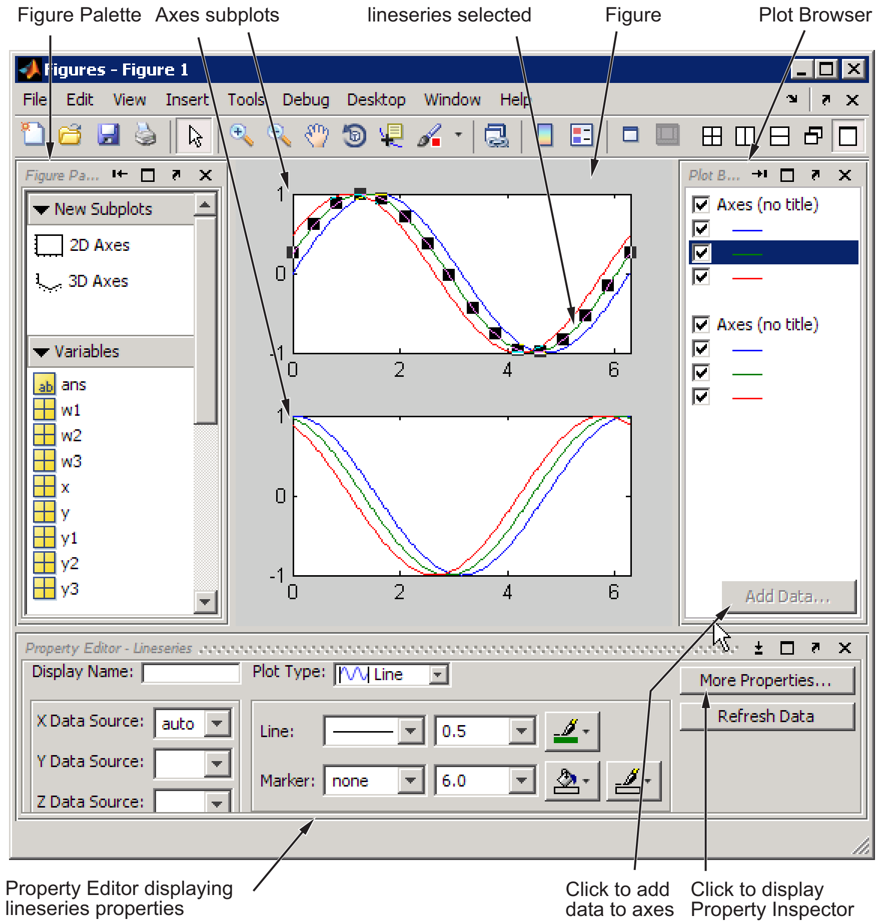
The example illustrated below shows the plotting tools attached to a figure containing two subplots of lineseries data. The code to produce the graphs is

```
x = 0:pi/100:2*pi;
y1 = sin(x);
y2 = sin(x+.25);
y3 = sin(x+.5);
subplot(2,1,1);
plot(x,y1,x,y2,x,y3);
axis tight;
w1 = cos(x);
w2 = cos(x+.25);
w3 = cos(x+.5);
subplot(2,1,2);
plot(x,w1,x,w2,x,w3);
axis tight;
```

You summon the plotting tools, either by selecting **Figure Palette**, **Plot Browser**, and **Property Editor** from the figure's **View** menu, or by typing

```
plottools
```

in the Command Window. Typing `plottools` or `plottools on` restores the configuration of tools the last time you were using them; use the View menu to show the ones you need and hide the ones you do not, and the mouse to dock and undock them. The default configuration of plotting tools is shown below. MATLAB preferences remember the current arrangement of plot tools — whether they are visible or not — each time you exit; if you want to revert to the default configuration you need to restore the arrangement shown below manually.



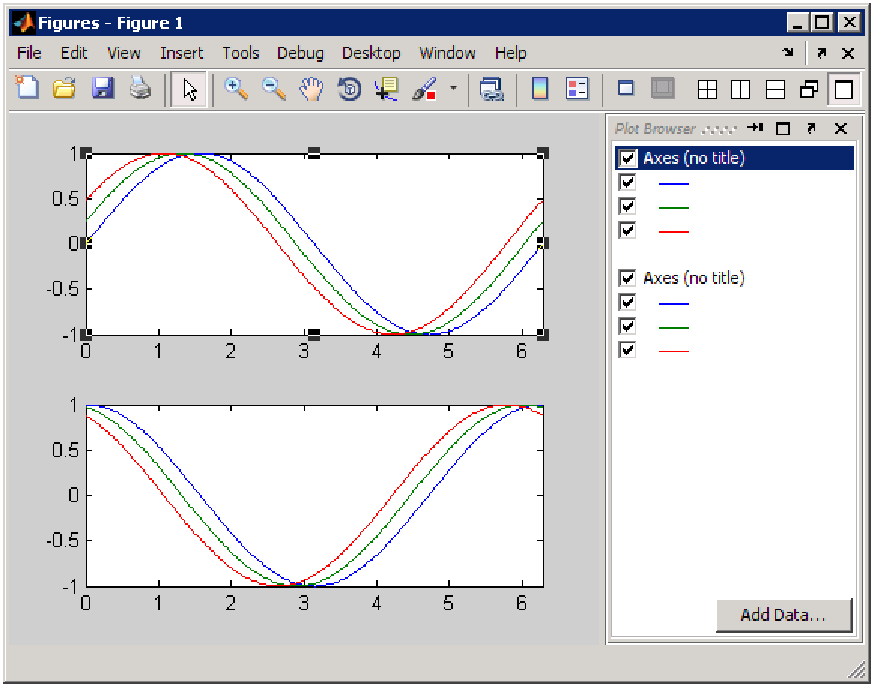
Managing Plotting Tools

Each of the plotting tools shown above can be docked or undocked from its figure, or dismissed by clicking the **x** at the right end of its titlebar. If you dismiss a tool and want it back again, you can raise it from the View menu or by typing one of several commands. For instance, if you had undocked, and then dismissed the figure palette, you could type either

`plottools( 'on', 'figurepalette')` or

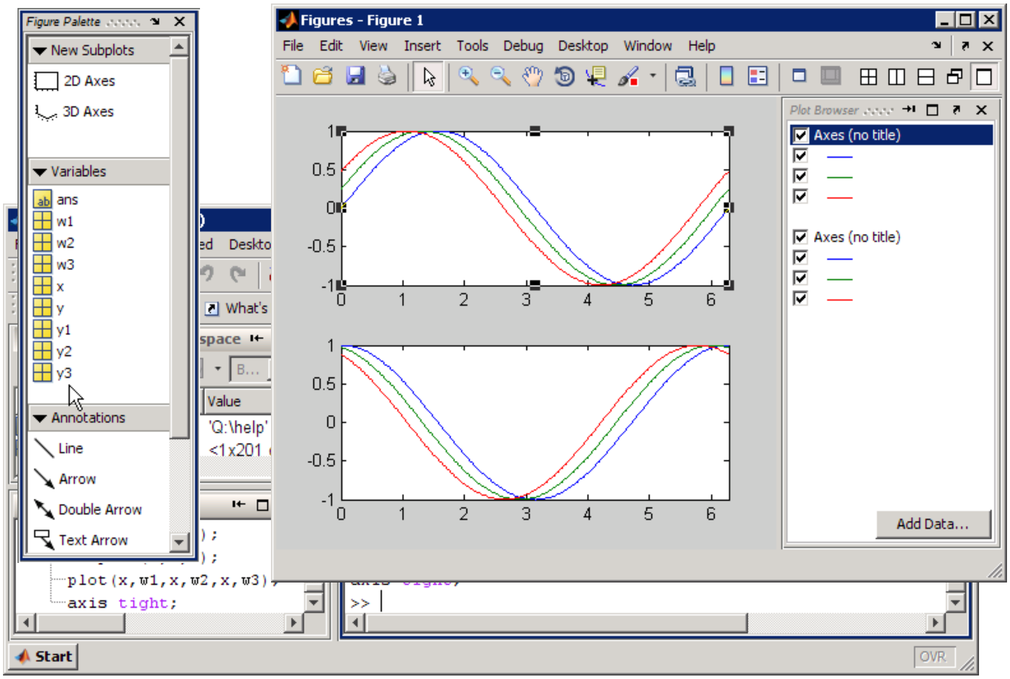
```
figurepalette
```

Your desktop configuration might then look like this:



Figures Groups. When you activate any plot tool for a figure (or dock the figure in the desktop), the figure becomes part of a Figures group. Figures groups are desktop containers that you can dock in your desktop. Individual figures are not dockable except within the Figures group container. If you create subsequent figures, they will also dock in the Figures group, where they can be panelled or overlapped for viewing. A row of tabs appears along the bottom, one for each figure in the group.

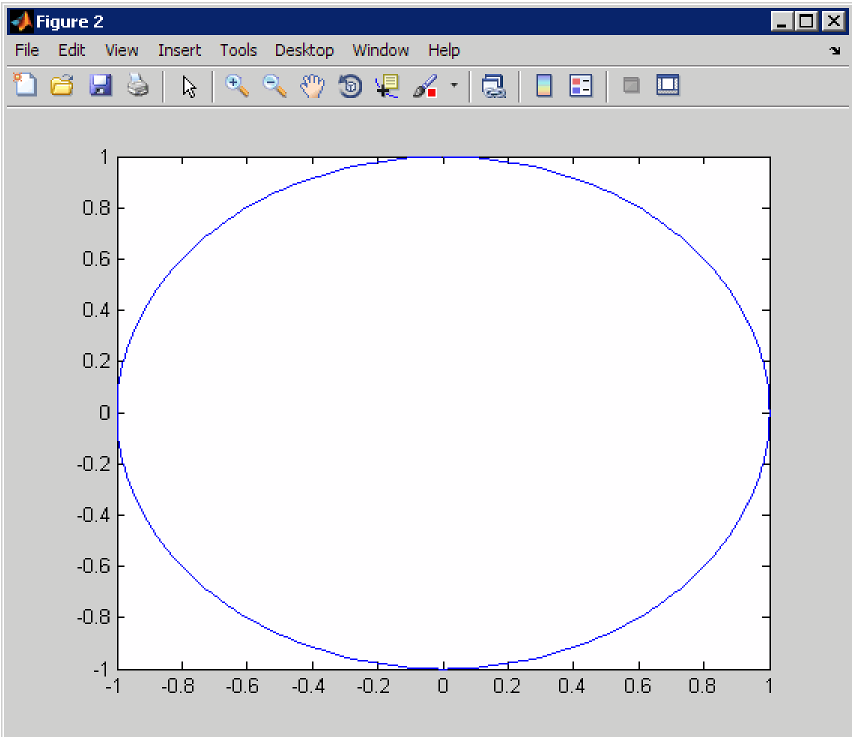
When you dock a plot tool in a figure group and then dock the figure group in the desktop, the tool is included in that section of the desktop as well, as the following illustration shows:

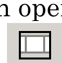


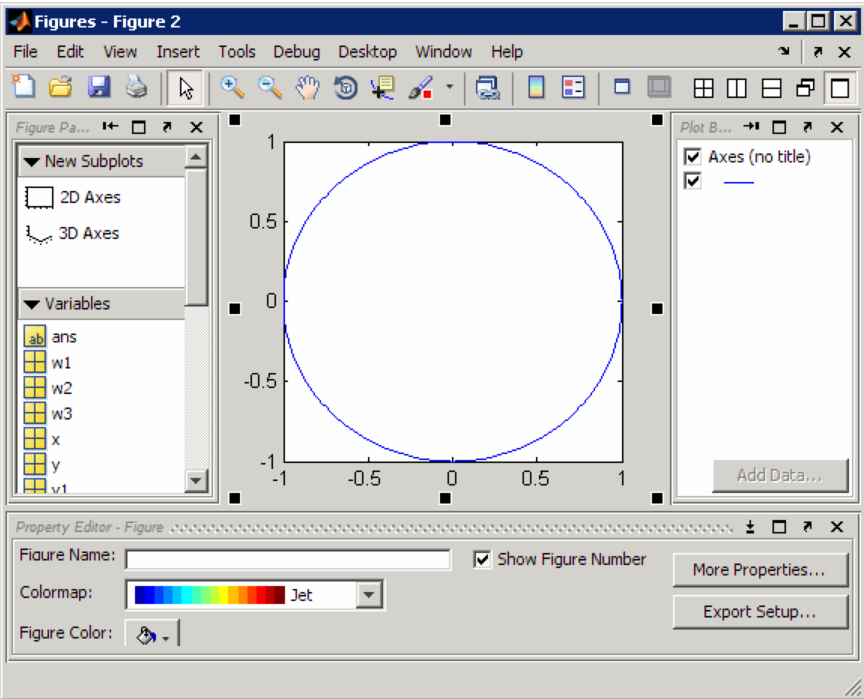
Working with Multiple Figures. When you create a new figure and plot into it, It is created without plotting tools enabled, even if another figure already has them open:

```
figure
plot(y1,w1)
```

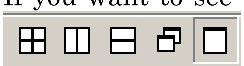
This generates a freestanding plot, like this:

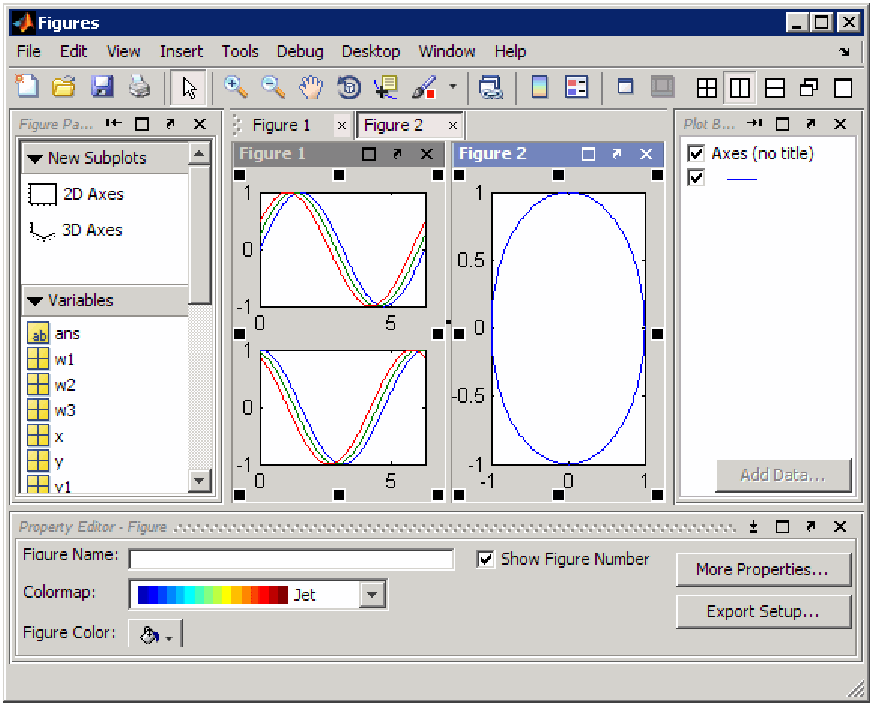


If you then open the plotting tools for the figure by clicking the Open Plotting Tools icon , the figure docks in the figure window:



The new figure might seem to disappear if the Figures window is hidden, but it will overlay the existing plot within that window (it does not replace it). You can switch between the two figures by clicking the tabs at the bottom of the figure area. Be aware that clicking the x on the right side of a figure's tab deletes that figure entirely, without asking for confirmation.

If you want to see both figures at once, use the Tiling Palette  at the upper right corner of the Figures window to arrange the figures. For example, clicking the Left/Right tiling tool lays out the two figures side by side:



As the above illustrations shows, plot objects can be selected in both figures, but only one figure has focus at a time.

The Figure Palette

The Figure Palette contains three panels. Select the panel you want to view by clicking the respective button, which twists down the panel and exposes its contents.

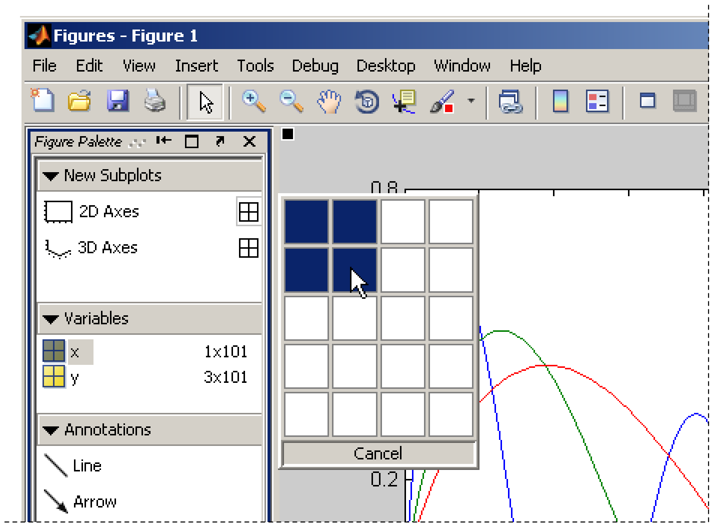
The Figure Palette enables you to perform the following tasks with these panels:

- **New Subplots** — Add 2-D or 3-D axes to the figure.
- **Variables** — Browse and plot workspace variables.
- **Annotations** — Add annotations to graphs.

Adding Subplot Axes

The **New Subplots** panel enables you to create a grid of either 2-D or 3-D axes. To display the selector, click the grid icon next to the axes type. A subplot selector grid appears.

As you move the cursor, squares darken to indicate the layout of axes that will be created if you release the mouse button. Click **Cancel** at the bottom of the grid to leave the figure unchanged.

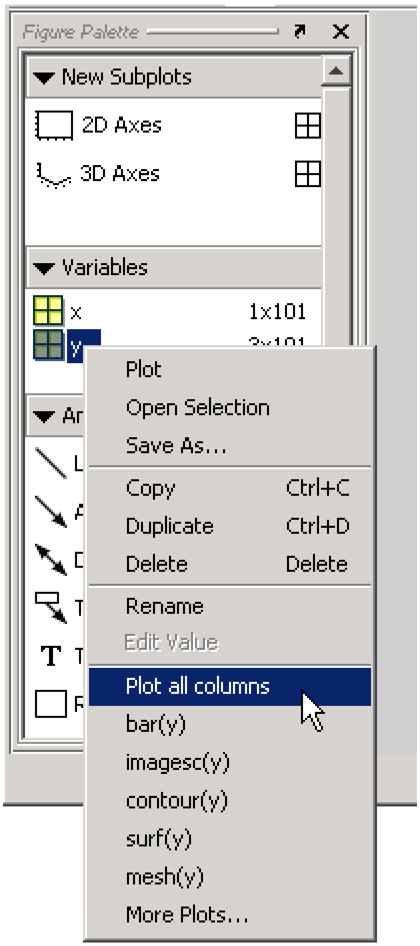


The picture above shows the **New Subplots** panel selected to display four equally sized axes in the figure. Existing axes resize as required to accommodate the new layout.

Plotting Workspace Variables

The **Variables** panel displays current workspace variables. Double-clicking a variable in this panel opens that variable in the Variable Editor. If you select a variable and right-click to display the context menu, you can select a graphics function to plot the variable.

For example, the following picture illustrates how to plot the columns of variable Z. This is equivalent to passing a matrix to the plot function.



The context menu contains a list of possible plot types based on the variable you select and also enables you to perform certain operation on the variable, such as opening it in the Variable Editor, saving, copying, and so on.

The context menu items may change when you select different variables because a particular variable might be incompatible some of the plot types.

Drag and Drop Plotting

You can also drag the variable directly into an axes, in which case MATLAB selects the first appropriate plot type for that variable. If there are multiple axes, you must first select the one you want to plot in and then drag the variable to that axes.

In the previous example, the variable `Z` would be plotted using the `plot` function if you were to drag it into an axes.

If the desired plotting function is not available from the context menu, you can select **More Plots** to display the Plot Catalog tool.

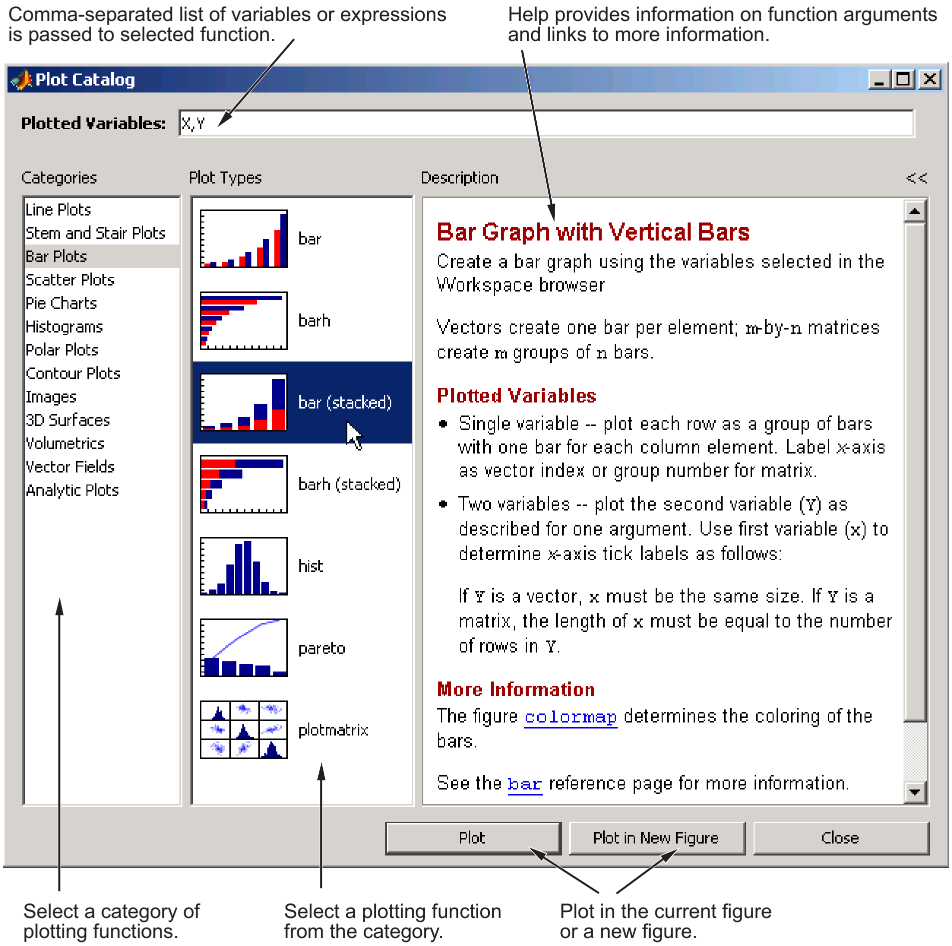
The Plot Catalog Tool

The Plot Catalog tool provides access to most of the MATLAB plotting functions. You can type any workspace variables or MATLAB expressions in the **Plotted Variables** field, which are then passed to the selected plotting function as arguments. Separate variables with a comma.

You can also enter function handles to pass to one of the “ez...” family of plotting functions from the Analytic Plots category.

Note The Plot Catalog does not prevent you from passing inappropriate or insufficient data to a plotting function; if the plot appears to be incorrect or if a plot fails to draw after you press **Plot** or **Plot in New Figure**, you should check the Command Window for warning and error messages.

The following picture shows the Plot Catalog tool and describes its components.



Adding Annotations to Graphs

The **Annotations** panel enables you to insert annotation objects into a plot. To add an object, first select the object you want to add, and then click and drag the mouse to position and size the object.

See "How to Annotate Graphs" on page 3-2 for more information about the various types of annotation objects.

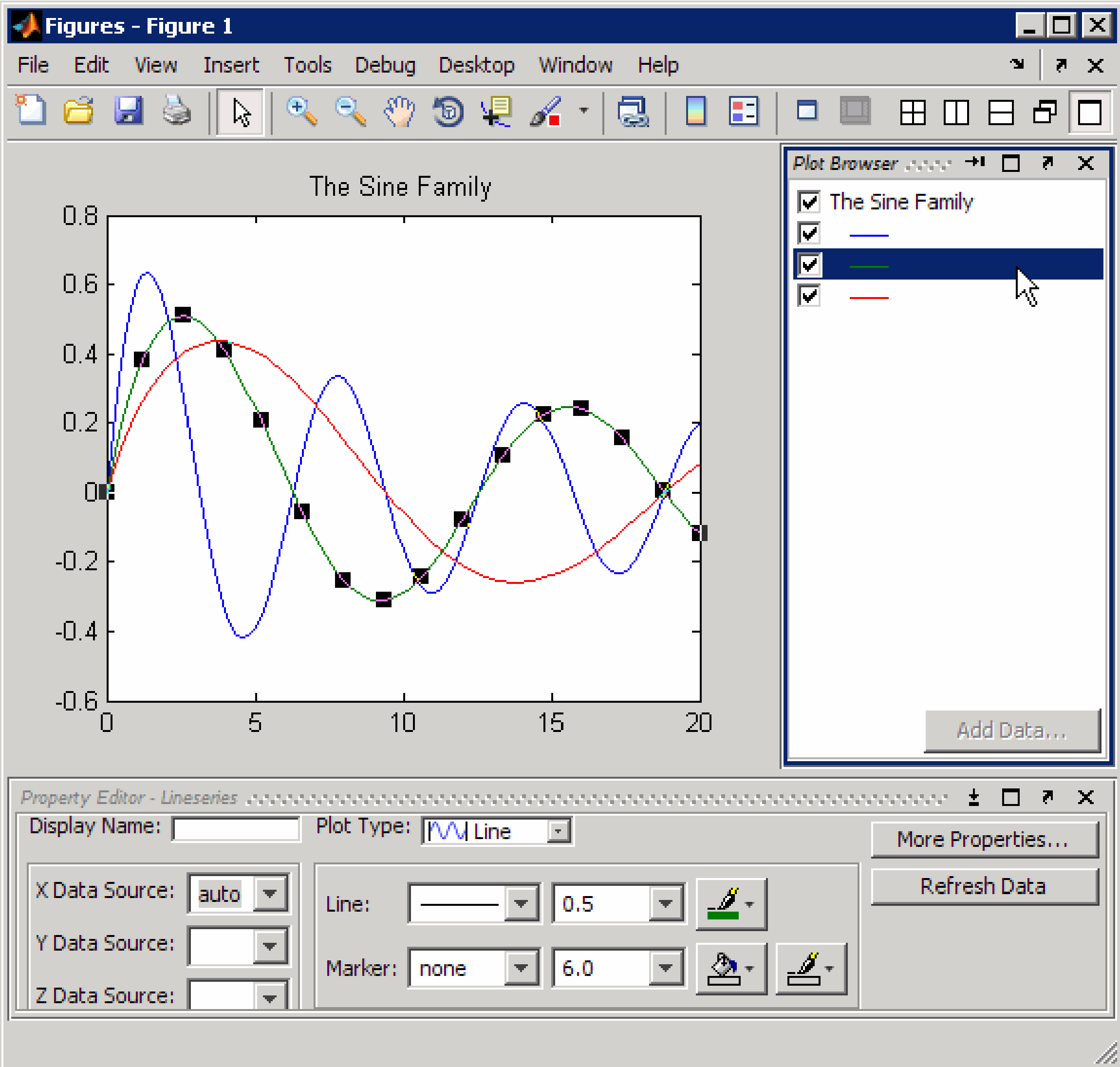
The Plot Browser

The Plot Browser provides a legend of all the graphs in the figure. It lists each axes and the objects (lines, surfaces, etc.) used to create the graph.

For example, suppose you plot an 11-by-11 matrix `z`. The `plot` function creates one line for each column in `z`.

```
plot(z,'DisplayName','z')
```

When you set the `DisplayName` property, the Plot Browser indicates which line corresponds to which column.

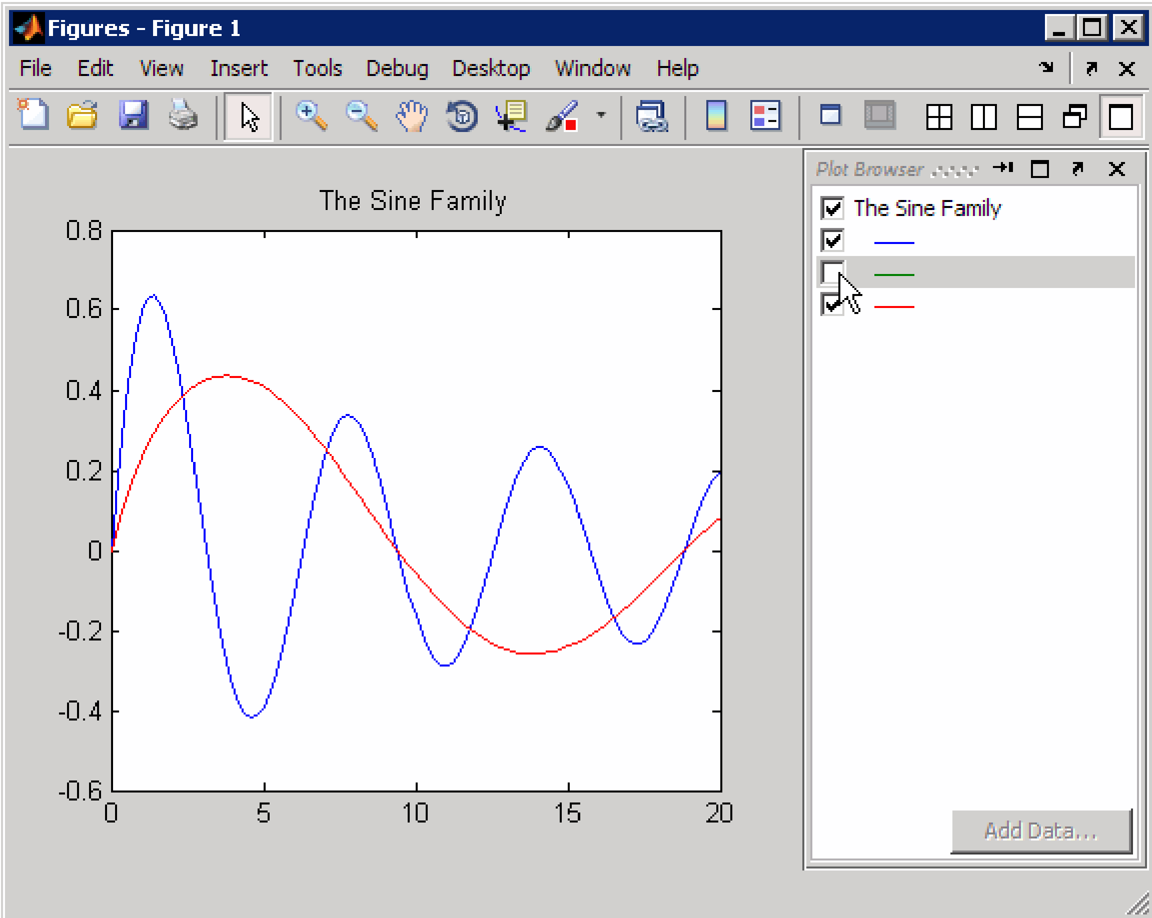


If you want to set the properties of an individual line, double-click on the line in the Plot Browser. Its properties are displayed in the Property Editor, which opens on the bottom of the figure.

You can select a line in the graph, and the corresponding entry in the Plot Browser is highlighted, enabling you to see which column in the variable produced the line.

Controlling Object Visibility

The check box next to each item in the Plot Browser controls the object's visibility. For example, suppose you want to plot only certain columns of data in z, perhaps the positive values. You can deselect the columns you do not want to display. The graph updates as you deselect each box and rescales the axes as required.



Deleting Objects

You can delete any selected item in the Plot Browser by selecting **Delete** from the right-click context menu.

Adding Data to Axes

The Plot Browser provides the mechanism by which you add data to axes. The procedure is as follows:

1 Select a 2-D or 3-D axes from the **New Subplots** subpanel.

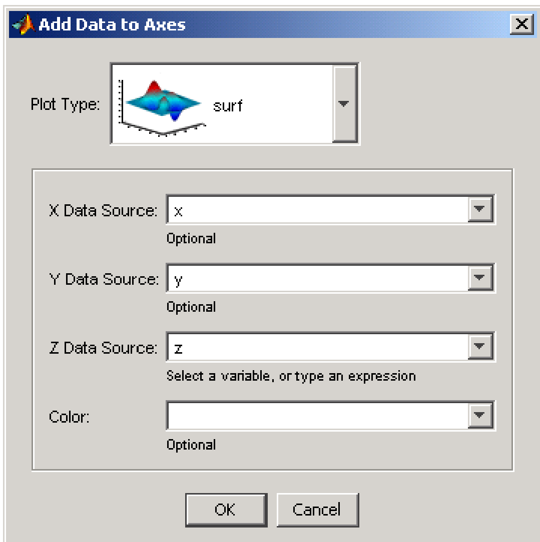
2 After creating the axes, select it in the Plot Browser panel to enable the **Add Data** button at the bottom of the panel.

3 Click the **Add Data** button to display the Add Data to Axes dialog.

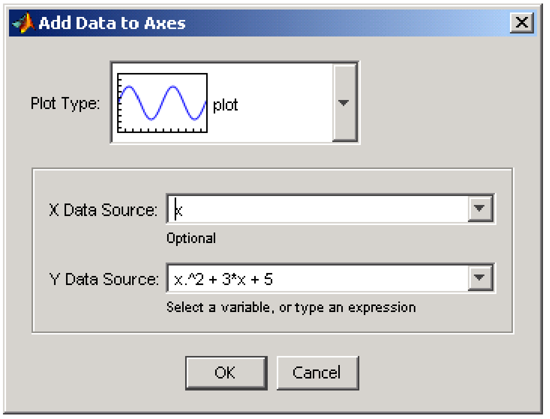
The Add Data to Axes dialog enables you to select a plot type and specify the workspace variables to pass to the plotting function. You can also specify a MATLAB expression, which is evaluated to produce the data to plot.

Selecting Workspace Variables to Create a Graph. Suppose you want to create a surface graph from three workspace variables defining the `XData`, `YData`, and `ZData` (see the `surf` function for more information on this type of graph).

In the workspace you have defined three variables, `x`, `y`, and `z`. To create the graph, configure the Add Data to Axes dialog as shown in the following picture.



Using a MATLAB Expression to Create a Graph. The following picture shows the Add Data to Axes dialog specifying a workspace variable `x` for the plot's x data and a MATLAB expression (`x.^2 + 3*x + 5`) for the y data.



You can use the default **X Data** value of `index` if you do not want to specify x data. In this case, the y data versus the index of the y data values are plotted, which is equivalent to calling the `plot` command with only one argument.

The Property Editor

The Property Editor enables you to access a subset of the selected object's properties. When no object is selected, the Property Editor displays the figure's properties.

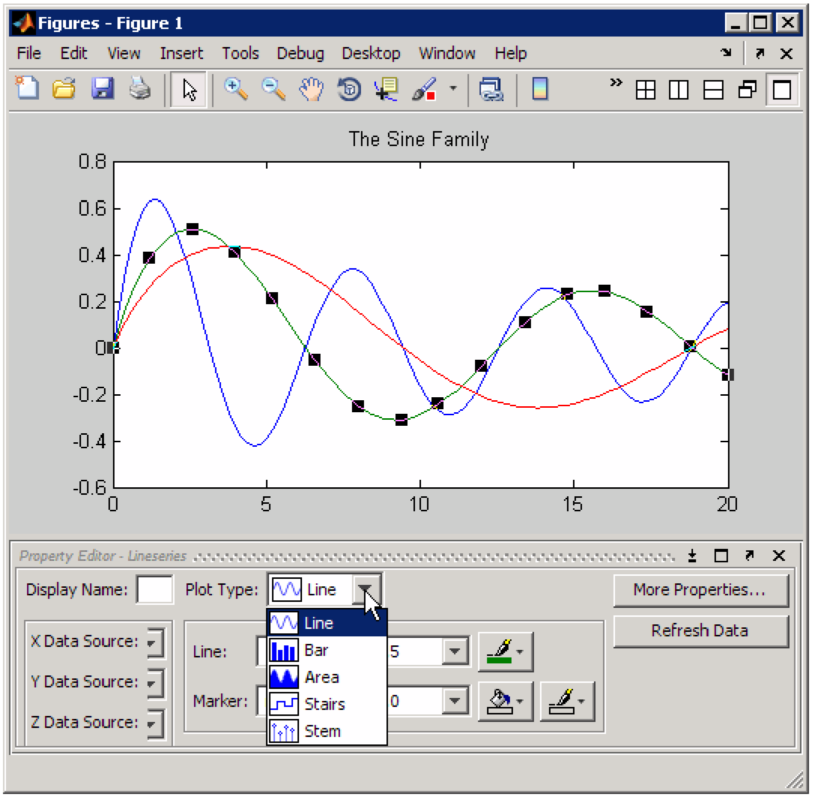
Ways to Display the Property Editor

There are a variety of ways to display the Property Editor:

- Double-click an object when plot edit mode is enabled.
- Select an object and right-click to display its context menu, then select **Properties.**
- Select **Property Editor** from the **View** menu.
- Use the `propertyeditor` command.

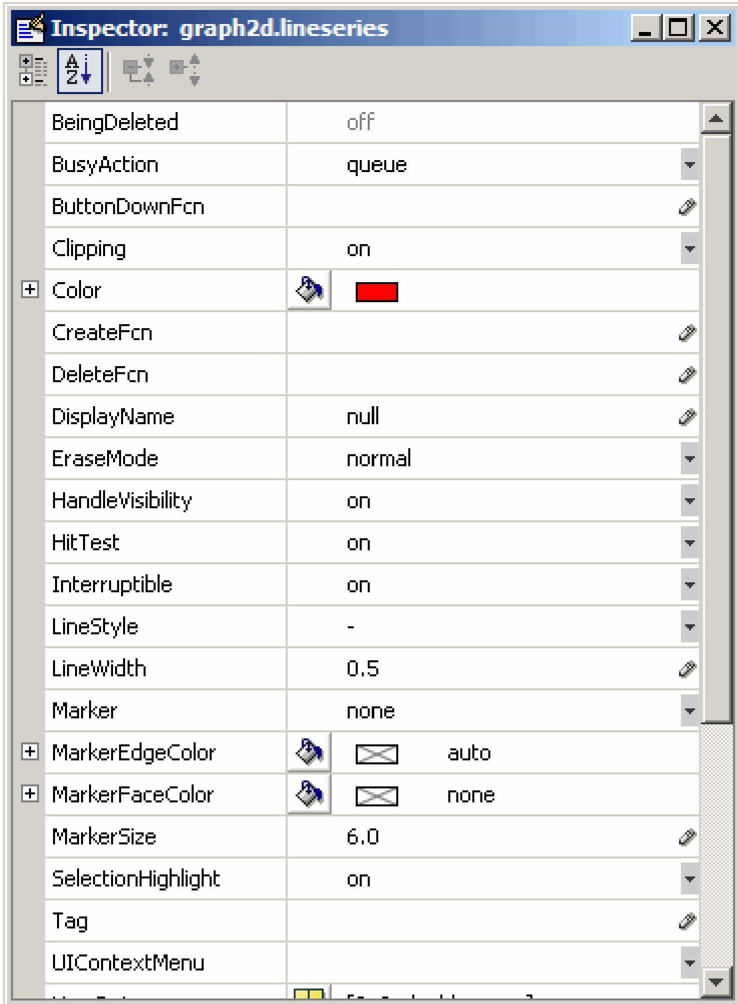
Changing Plot Types

You can use the property editor to change the type of plot used to display data. For example, you can change the following line graph to a stem, stairs, area, or bar graph by changing the **Plot Type** field.



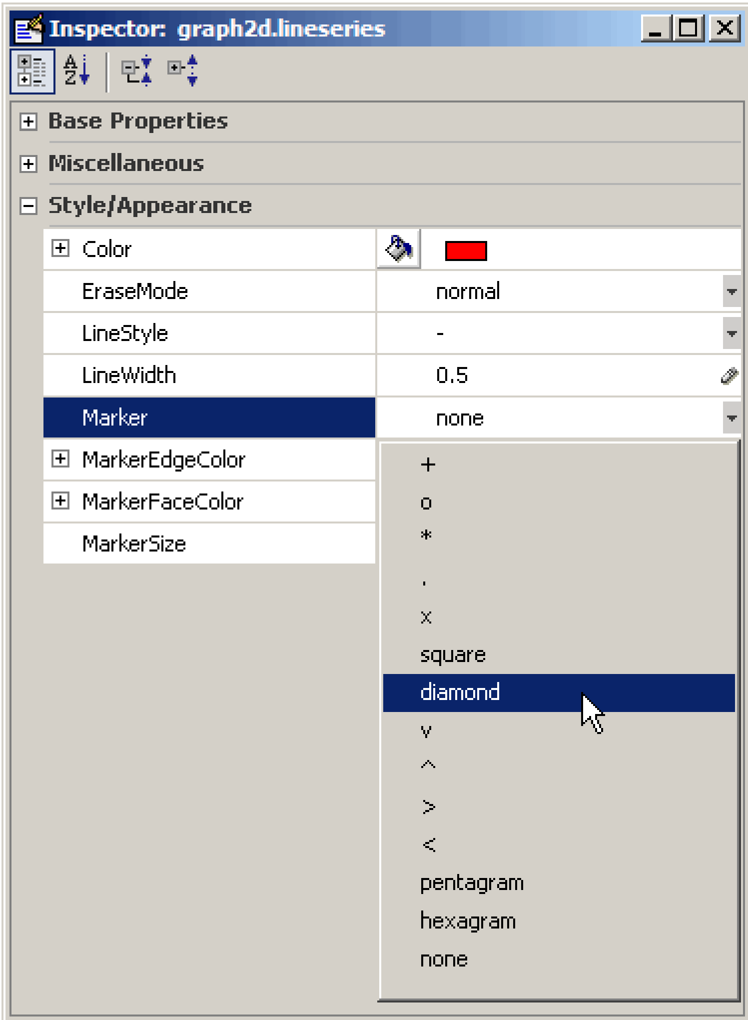
Accessing Object Properties with the Property Inspector

The Property Editor enables you to change the most commonly used object properties. To access more object properties, use the Property Inspector. Open the Property Inspector by clicking the **More Properties** button on the Property Editor or by typing `inspect` in the Command Window. The following picture shows the Property Inspector displaying the properties of the same lineseries object as that in the previous picture.



The default view of properties is an alphabetic list; you can change to a tree view by clicking the icon in the upper left corner containing plus marks. Click the "AZ" icon to its right to return to an alphabetized list view. Properties that contain fields, such as RGB color components, have a plus mark to their left you can click to expose the individual values. You can change properties that have enumerated values, such as `Marker`, via drop-down lists (downward-pointing triangles on the right edge of the inspector window). The

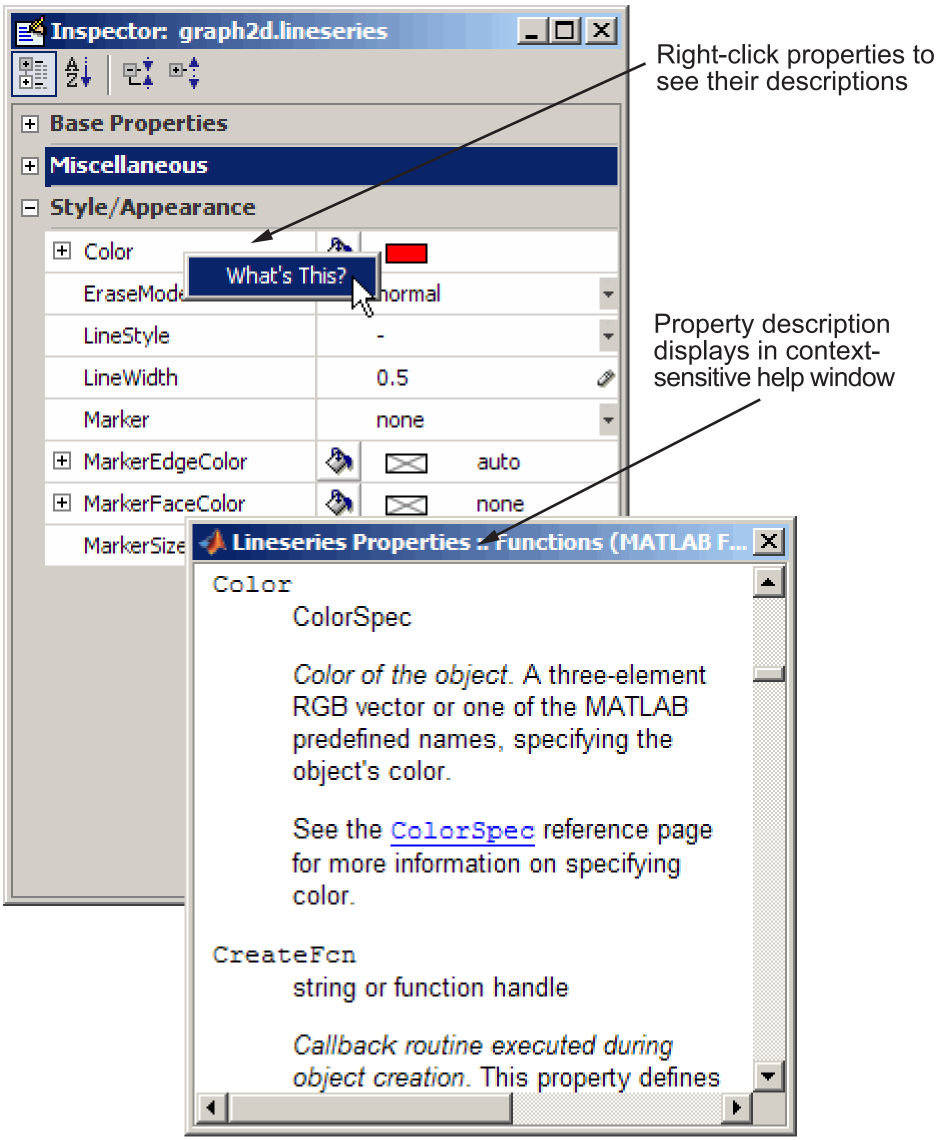
following figure shows the Marker property being set to diamond using the Property Inspector (note the tree view in which other groups of properties have been collapsed):



There are a few properties of objects that the Property Inspector does not show, for example Parent and Children. For complete descriptions of the properties of graphics objects, use the Handle Graphics Property Browser.

Getting Help for Object Properties

If you are not sure what a property does or what values it can take on, you can get a description of it from the Property Inspector. To do so, right-click on the name or values of a property and select **What's This** from the popup menu; a Help window opens displaying the property reference page for the current object, scrolled to the property you clicked. The following picture shows how this works:


Inspector: graph2d.lineseries
Base Properties
Miscellaneous
Style/Appearance
Color
What's This?
EraseMode
normal
LineStyle
-
LineWidth
0.5
Marker
none
MarkerEdgeColor
auto
MarkerFaceColor
none
MarkerSize
Right-click properties to see their descriptions
Property description displays in context-sensitive help window
Lineseries Properties :: Functions (MATLAB F...
Color
ColorSpec
Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color.
See the ColorSpec reference page for more information on specifying color.
CreateFcn
string or function handle
Callback routine executed during object creation. This property defines

Accessing Objects You Cannot Click

If you want to access the properties of light or `uicontextmenus` objects, you need to get the handle using MATLAB commands, because you cannot click on these objects.

For example, to get the handles of all light objects in the current axes, use `findobj`.

```
h = findobj(gca,'Type','light');
```

Then use the `inspect` command to display the Property Inspector.

```
inspect(h)    % Inspect all light objects
inspect(h(1)) % Inspect the first light object in list
```

Example — Working with Plotting Tools

In this section...
“Identifying Workspace Data to Plot” on page 1-35
“Adding a Subplot” on page 1-38

Identifying Workspace Data to Plot

This example illustrates how to use the plotting tools to graph a workspace variable versus an expression typed into the **Add Data to Axes** dialog.

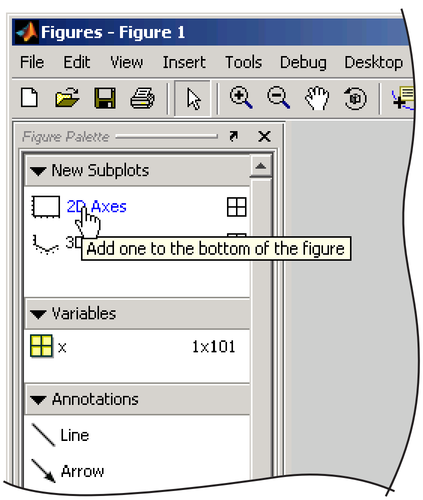
Create a variable in the workspace,

```
x = -2*pi:pi/25:2*pi;
```

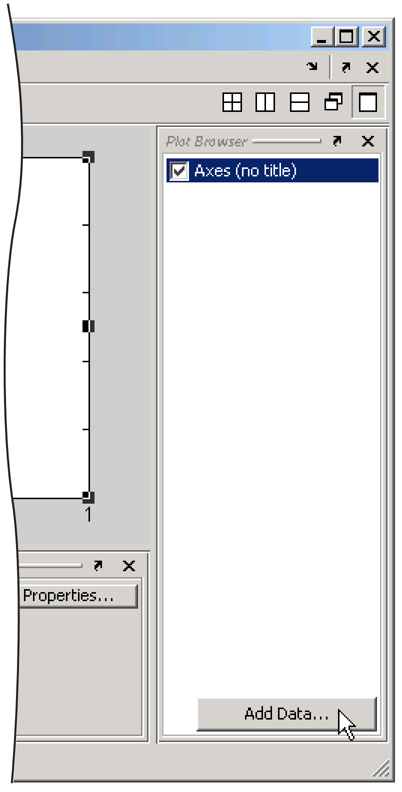
Use the `plottools` command to create a figure group with the plotting tools attached.

```
plottools
```

Click **2D Axes** in the **New Subplot** panel of the Figure Palette.

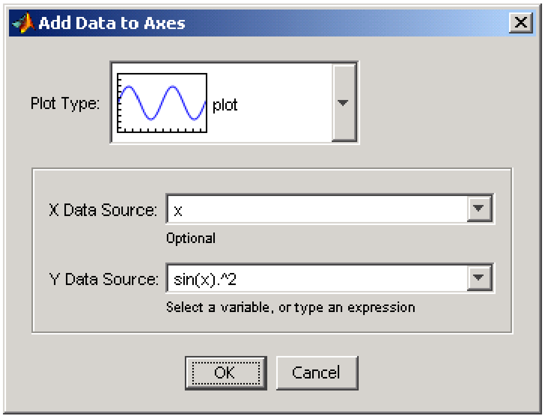


Once the axes appears, the **Add Data** button on the Plot Browser is activated.



Click this button to display the Add Data to Axes dialog. When the Add Data to Axes dialog is displayed, enter the following values:

- Select plot as the **Plot Type**.
- Set **X Data Source** to `x`.
- Set **Y Data Source** to `sin(x).^2`.
- Click **OK** to plot this data.

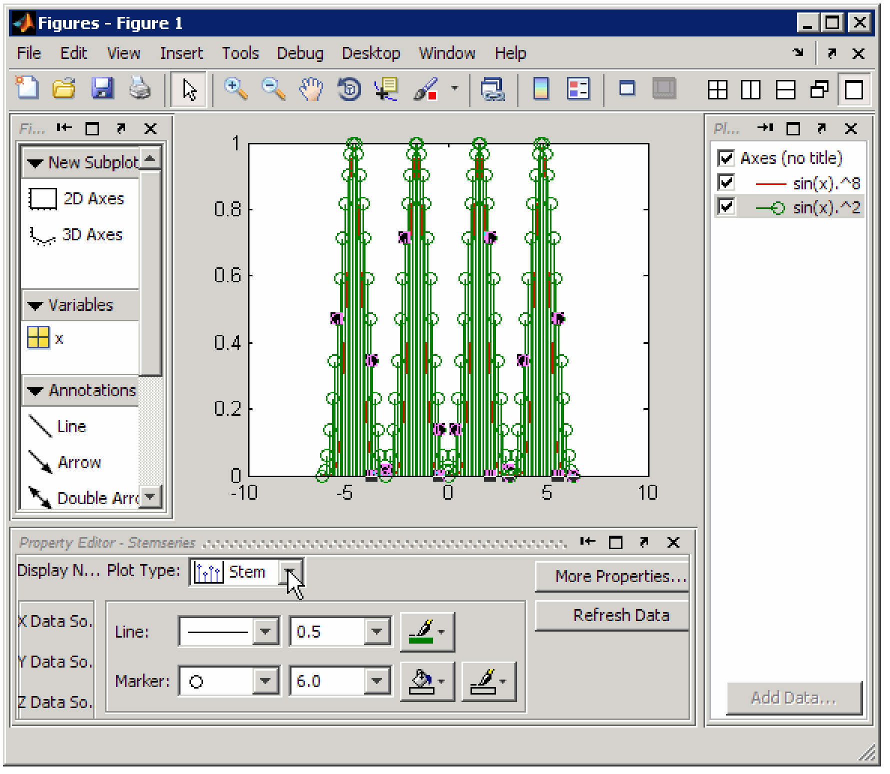


A plot of `sin(x).^2` versus `x` appears.

Now add another plot to the same axes. Click **Add Data** again and specify the data to plot:

- Set **X Data Source** to `x`.
- Set **Y Data Source** set to `sin(x).^8`.
- Click **OK** to plot this data.

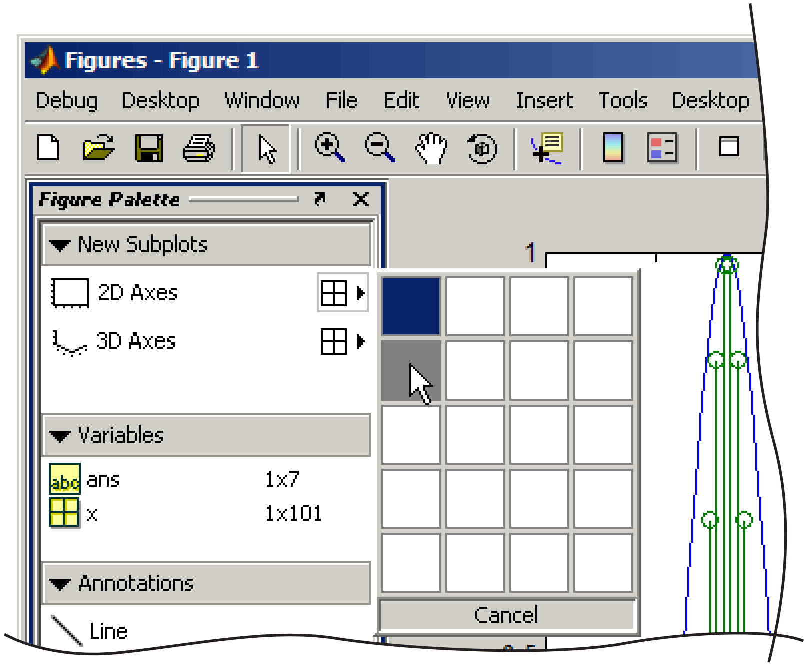
Select the last plot (the green line) and set the `Plot Type` in the Property Editor to `Stem`. The plot should now look like the following picture.



Adding a Subplot

Add a second axes below the current axes using the **New Subplots** panel. Click the right-facing arrowhead next to **2D Axes** and move the mouse to darken two squares, one on top of the other.

This creates a subplot axes below the existing axes. The existing axes resize so both fit in the figure.



In the inserted axes, select its entry in the Plot Browser and then click **Add Data**.

When the Add Data to Axes dialog is displayed, enter the following values:

- Set **X Data Source** to `x`.
- Set **Y Data Source** to `sin(x).^3`.
- Click **OK** to plot this data.

Now add another plot overlaid on the first by clicking **Add Data** again and specify the data to plot:

- Set **X Data Source** to `x`.

- Set **Y Data Source** to `sin(x).^9`.
- Click **OK** to plot this data.

Select the plot labeled `sin(x).^9` under the second axes in the Plot Browser. Set the `Plot Type` in the Property Editor to `Area`.

Setting Axis Limits

Adjust the *x*-axis in both axes using the Property Editor.

- Select the first axes in the Plot Browser.
- Change **X Limits** to -7 and 7.

Repeat these steps for the second axes.

Adding Titles and Labels

Select the first axes in the Plot Browser and set the following properties in the Property Editor:

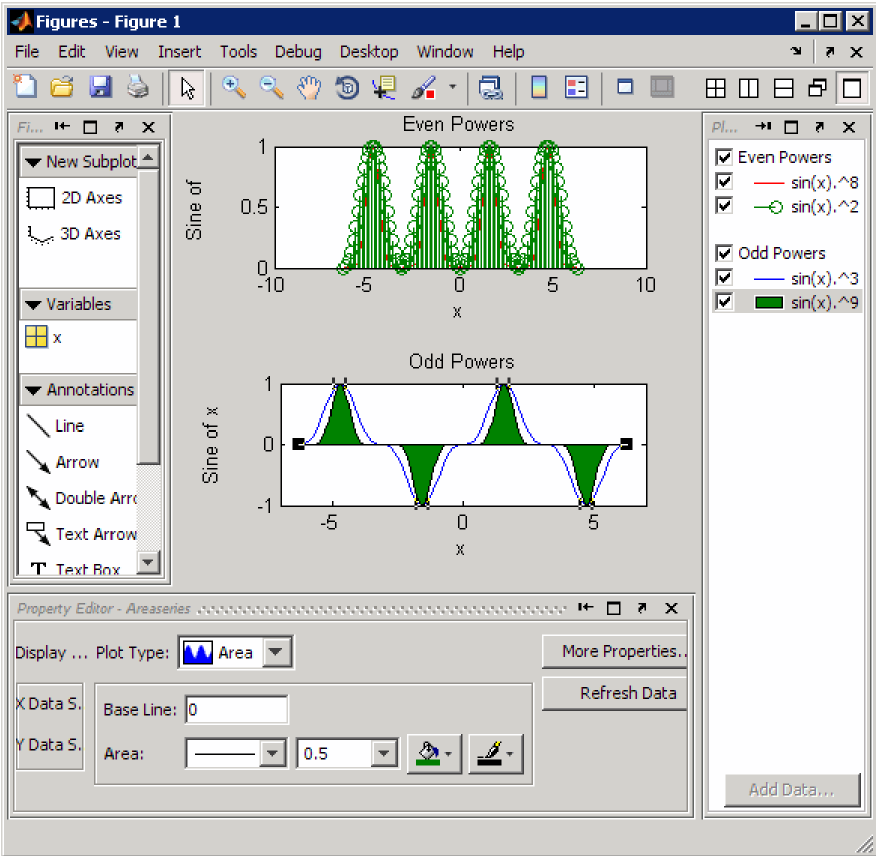
- Set **Title** to `Even Powers`.
- Set **X Label** to `X`.
- Click the **Y Axis** tab and set **Y Label** to `Sine of X`.

Select the second axes in the Plot Browser and set the following properties in the properties panel:

- Set **Title** to `Odd Powers`.
- Set **X Label** to `X`.
- On the **Y Axis** tab, set **Axis label** to `Sine of X`.

The Plot Browser now reflects the new axes names.

The following picture shows the result of these steps.

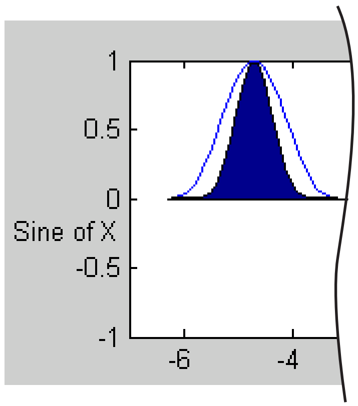


Select the text of the *y*-axis label on the first axes (now labeled `Even Powers` in the Plot Browser) and click the **More Properties** button on the Property Editor. Set the `Rotation` property to `0` and reposition the text by hand.

To make more space for the *y*-axis label, which is now in a horizontal position, select the axes and move it to the right with the mouse.

Repeat this process for the second axes (labeled `Odd Powers` in the Plot Browser).

The repositioned text label now looks like the following picture.



Note You can always undo your last change to the graph by selecting **Undo** from the **Edit** menu.

Example — Plotting from the Figure Palette

In this section...

Using the Plot Catalog

This example shows how to use the Figure Palette to select variables to plot. Suppose you have three variables in your workspace (x, y, z) created by the following statements:

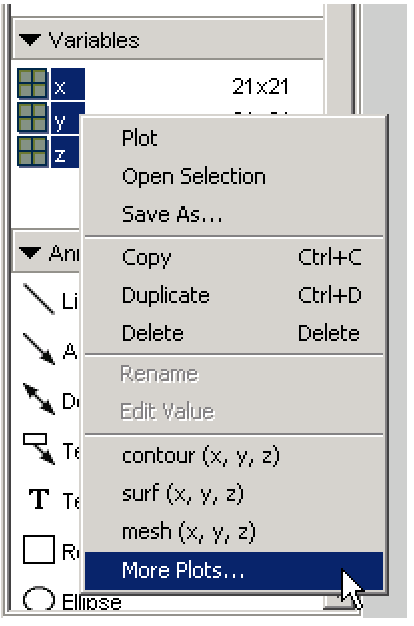
```
[x,y] = meshgrid([-2:.2:2]);
z = x.*exp(-x.^2-y.^2);
```

You decide to visualize this data as a surface/contour plot (as produced by the `surfc` function).

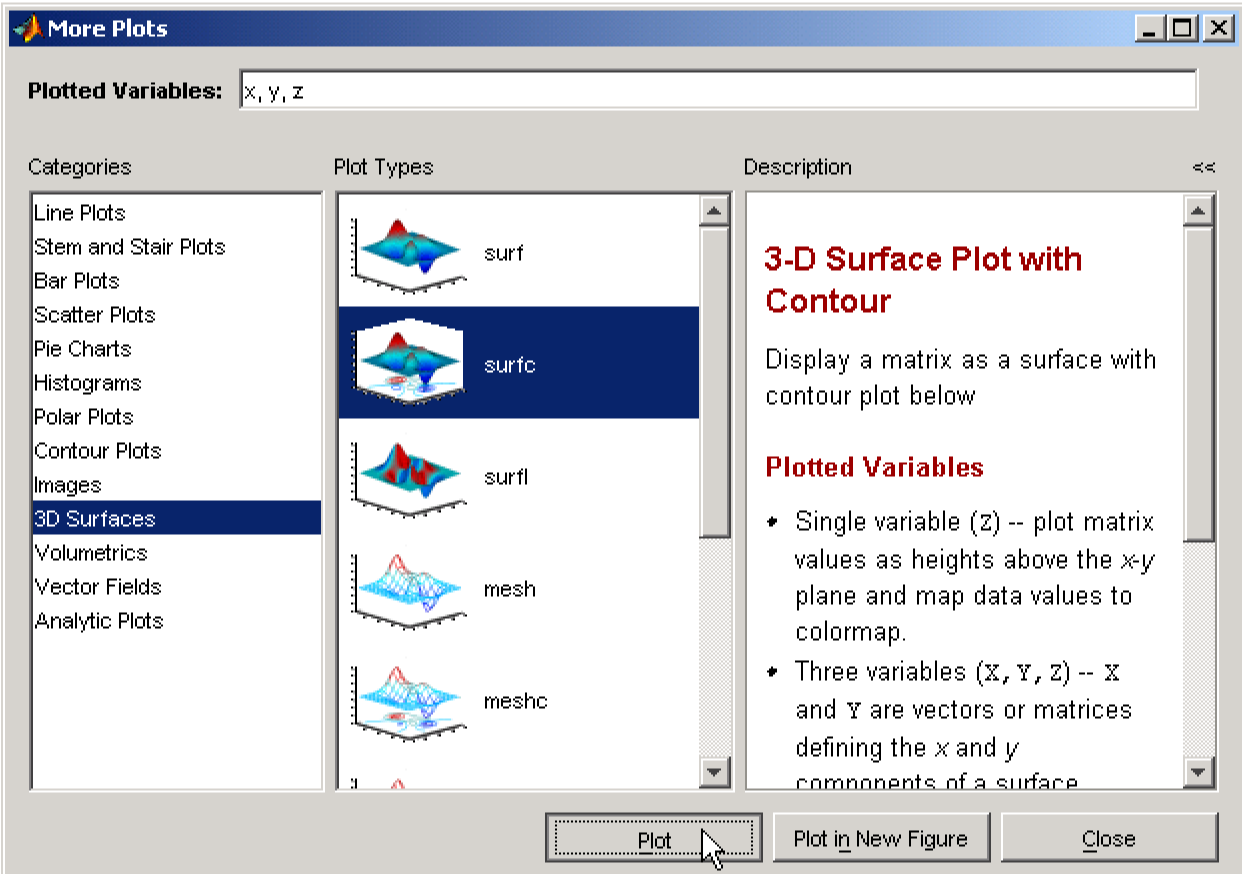
The first step is to display a figure with the Figure Palette tool attached. You can do this with the `figurepalette` command.

```
figurepalette
```

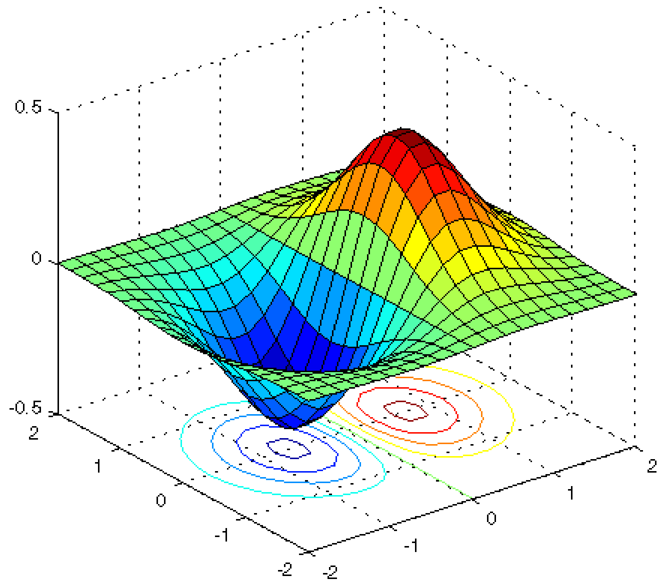
Expand the **Variables** panel and **shift-click** (for multiple selection) on the three variables you want to pass the plotting function. Since `surfc` is not in the list, select **More Plots**.


Variables
x
21x21
y
z
Plot
Open Selection
Save As...
Copy Ctrl+C
Duplicate Ctrl+D
Delete Delete
Rename
Edit Value
contour (x, y, z)
surf (x, y, z)
mesh (x, y, z)
More Plots...
Ellipse

From the Plot Catalog tool, select the **3D Surfaces** in the **Categories** column and **surfc** as the **Plot Type**, as shown in the following picture. To create the plot, click the **Plot** button.



The following graph results.



Plotting Expressions

You can enter MATLAB expressions in the Plot Catalog tool, as well as variables. For example, suppose you have created the following variables in the workspace.

```
t = 0:.01:20;
alpha =.055;
```

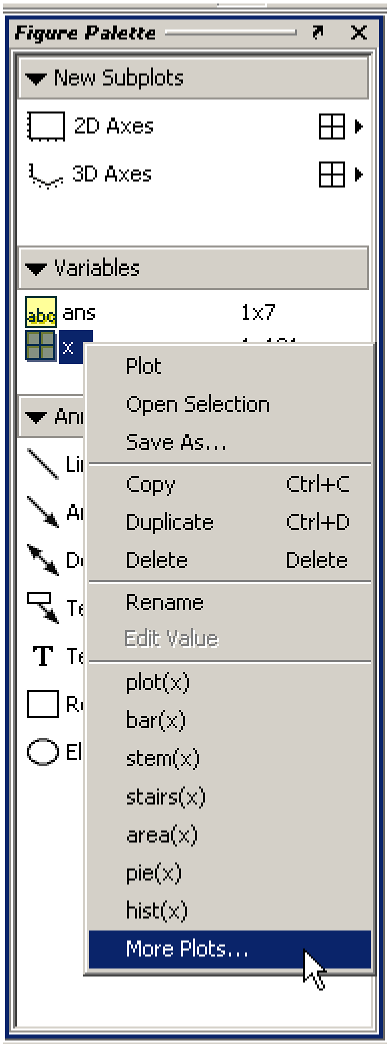
and you want to plot `t` versus this expression:

```
exp(-alpha*t).*sin(.5*t)
```

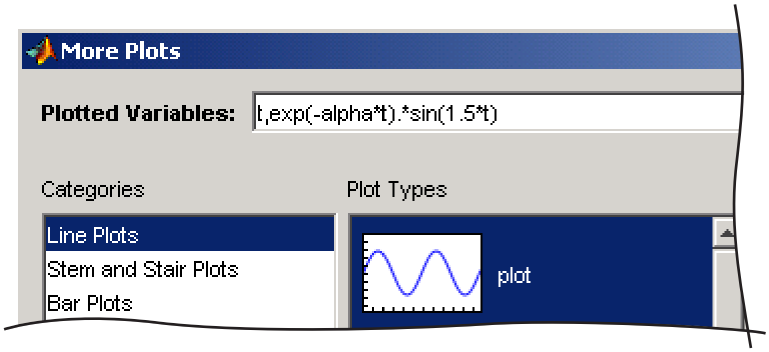
The first step is to display a figure with the Figure Palette tool attached. You can do this with the `figurepalette` command.

```
figurepalette
```

First select the variable t and right-click to display the context menu. Select **More Plots**.



When the Plot Catalog tool is displayed, add the expression to the **Plotted Variables** text field. You can reference the variable `alpha` because you created it in the base workspace. See MATLAB Workspace for information about variables in the MATLAB Workspace.



Click the **Plot** button to create the graph. The previous step is the equivalent of issuing the following command:

```
plot(t,exp(-alpha*t).*sin(.5*t))
```

Example — Specifying a Data Source

In this section...

Creating the Graph

First define two variables by issuing these statements in the command window.

```
t = 0:.01:20;
alpha =.055;
```

Next plot `t` versus the expression `exp(-alpha*t).*sin(.5*t)` using the `plot` function or the plot tools.

```
plot(t,exp(-alpha*t).*sin(.5*t))
```

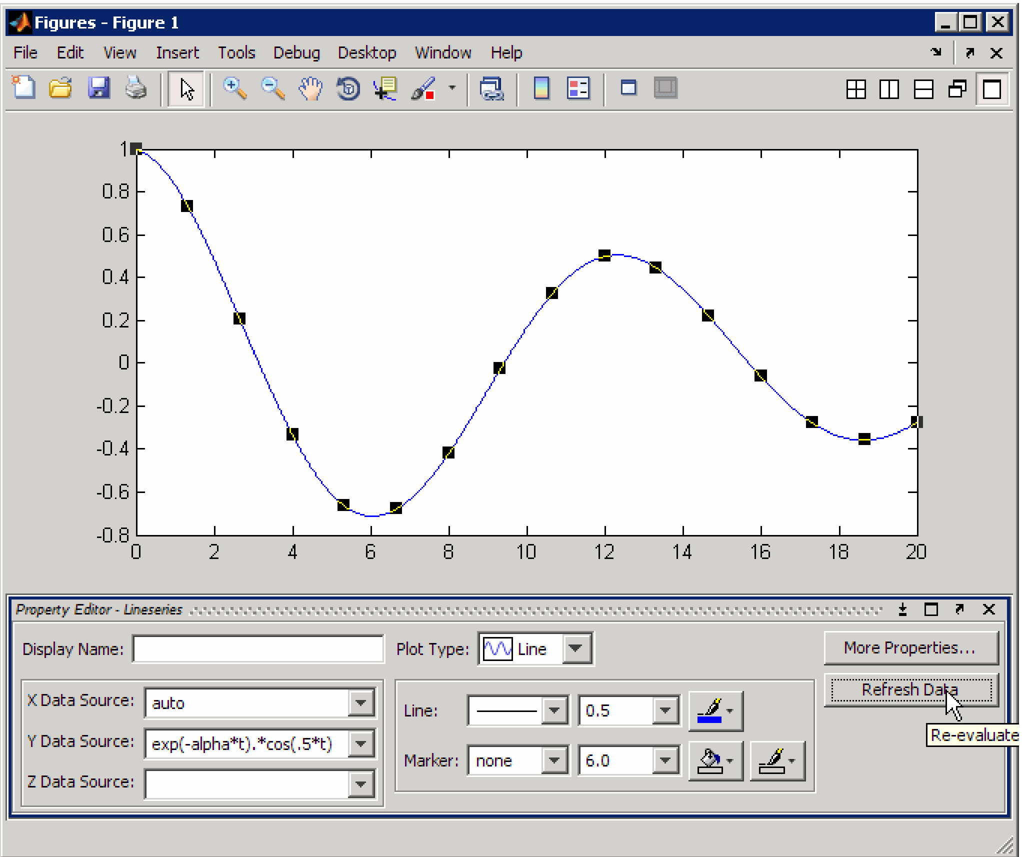
Varying the Data Source

Plot objects have properties that enable you to specify the source of the data that defines the object. For example, you can specify a workspace variable name or a MATLAB expression as the value of the `XDataSource`, `YDataSource`, or `ZDataSource` property for a line in a plot (i.e., a lineseries object). You can then use the Property Editor to change the variable name or alter the expression, and the plot is updated to reflect the change.

After creating the graph, you can use the Property Editor to couple the plotted line to the MATLAB expression.

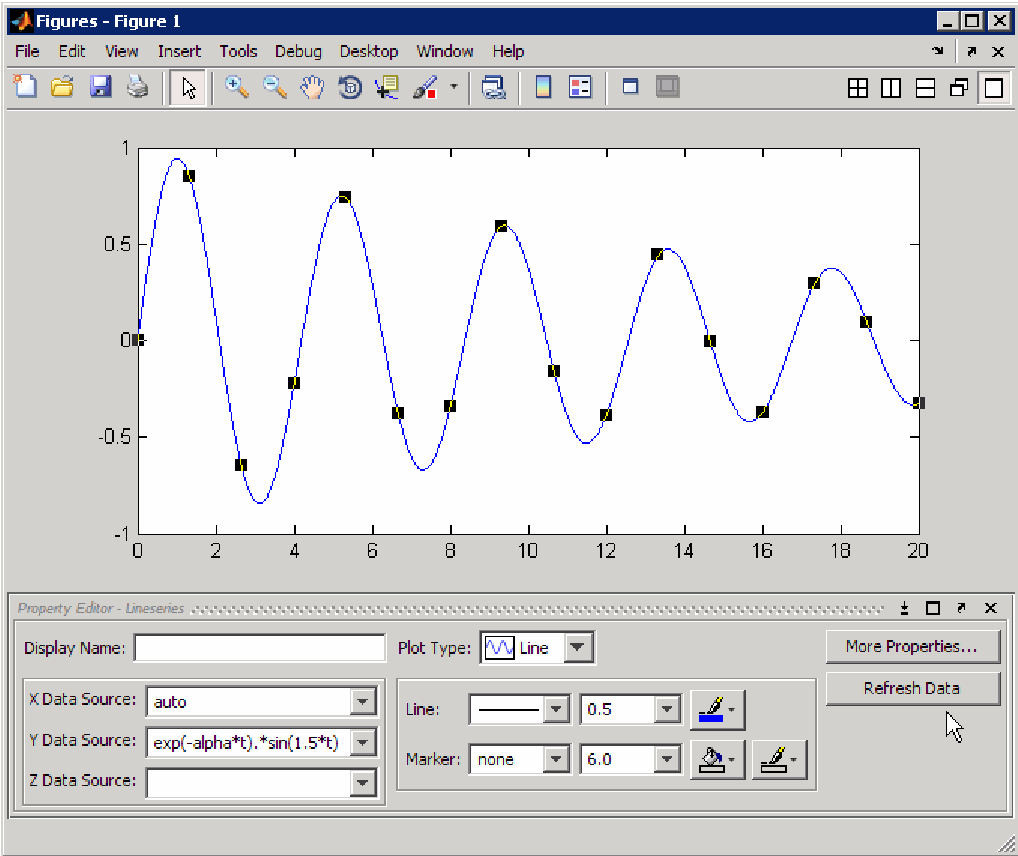
1. Double-click on the plotted line to display its property panel.

2. Enter the MATLAB expression `exp(-alpha*t).*cos(.5*t)` in the **Y Data Source** text field.



You can now modify the expression in the **Y Data Source** text field and observe how the graph changes. After changing the text, click the **Refresh Data** button to update the data.

In the following picture, `alpha` is no longer negated, so the function grows instead of decays. Also the period has been shortened by changing `sin(.5*t)` to `sin(1.5*t)`.



Data Sources for Multiobject Graphs

Suppose you create a line graph from matrix data. For example,

```
z = peaks;
h = plot(z,'YDataSource','z');
```

Because there is one lineseries object for each column of z, the following is true.

The data source for h(1) is z(:,1).

The data source for `h(2)` is `z(:,2)`.

...

The data source for `h(n)` is `z(:,n)`.

Example — Generating M-Code to Reproduce a Graph

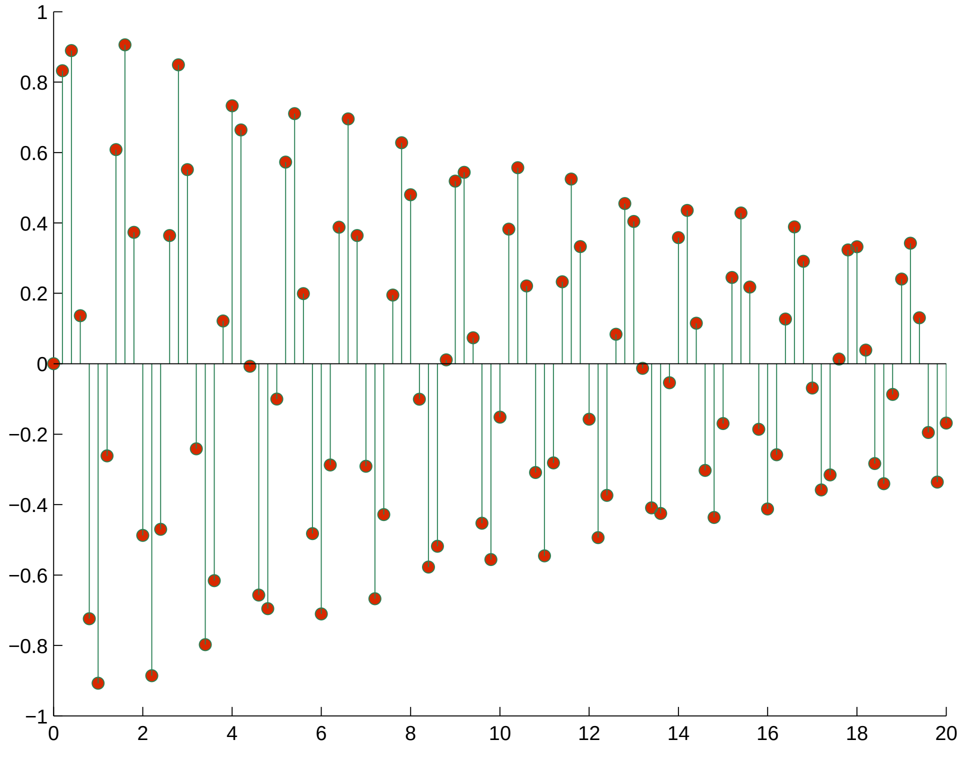
In this section...

Create a Stem Plot and Generate Code for It

Suppose you have created the following graph.

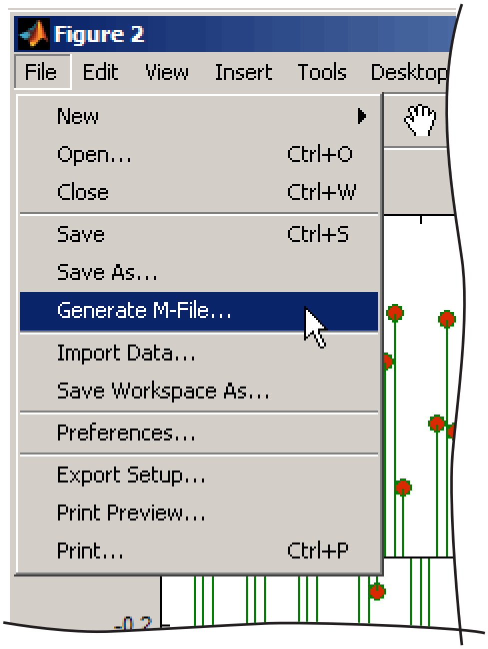
```
t = 0:.2:20;
alpha =.055;
stem(t,exp(-alpha*t).*sin(5*t))
```

Use the Property Editor to modify the graph. Select the stemseries and change the marker fill color to dark red, and marker edge color and line color to dark green. Remove the axes box, and change the font size for the axes labels to 8 to look like the following picture:



You can generate code to reproduce this graph by selecting **Generate M-File** from the **Figure** menu. MATLAB code generation software composes a function that recreates the graph and opens the generated M-File in the editor.

This feature is particularly useful for capturing property settings and other modifications you make using the plot tools GUI.



The M-file appears in an editor window and consists of the following code:

```
function createfigure(X1, Y1)
%CREATEFIGURE(X1,Y1)
%  X1:  stem x
%  Y1:  stem y

%  Auto-generated by MATLAB on 24-May-2006 14:23:45

% Create figure
figure1 = figure('Color',[1 1 1]);

% Create axes
axes('Parent',figure1,'FontSize',8);
hold('all');

% Create stem
stem(X1,Y1,'MarkerFaceColor',[0.8471 0.1608 0],...
    'MarkerEdgeColor',[0.1686 0.5059 0.3373],...
```

```
    'Color',[0 0.498 0]);
```

You must save the M-file before exiting MATLAB if you want to use it in future sessions.

Data Arguments

Generated functions do not store the data necessary to recreate the graph. You must supply the data arguments `t` as `X1` and `exp(-alpha*t).*sin(5*t)` as `Y1` to the function to recreate your graph. Of course, you can call the generated function with other argument pairs too.

Limitations

Attempting to generate code for graphs containing a large number of graphics objects (e.g., greater than 20 plotted lines) might be impractical.

Editing Plots

In this section...

Why Edit Plots?

MATLAB graphs are formatted to provide readability, setting the scale of axes, including tick marks on the axes, and using color and line style to distinguish the plots in the graph. However, if you are creating presentation graphics, you might want to change this default formatting or add descriptive labels, titles, legends, and other annotations to help explain your data.

You can edit the plots you create two ways:

- Using the mouse to select and edit objects interactively
- Using MATLAB functions at the command line or in an M-file

Interactive Plot Editing

If you enable plot editing mode in the MATLAB figure window, you can perform point-and-click editing of your graph. In this mode, you can modify the appearance of a graphics object by double-clicking on the object and changing the values of its properties. You access the properties through a graphical user interface called the Property Editor.

For more information about interactive editing, see "Working in Plot Edit Mode" on page 1-59.

For information about editing object properties in plot editing mode, see "The Property Editor" on page 1-28.

Using Functions to Edit Graphs

If you prefer to work from the MATLAB command line or if you are creating an M-file, you can use MATLAB commands to edit the graphs you create.

Taking advantage of the MATLAB Handle Graphics® system, you can use the `set` and `get` commands to change the properties of the objects in a graph.

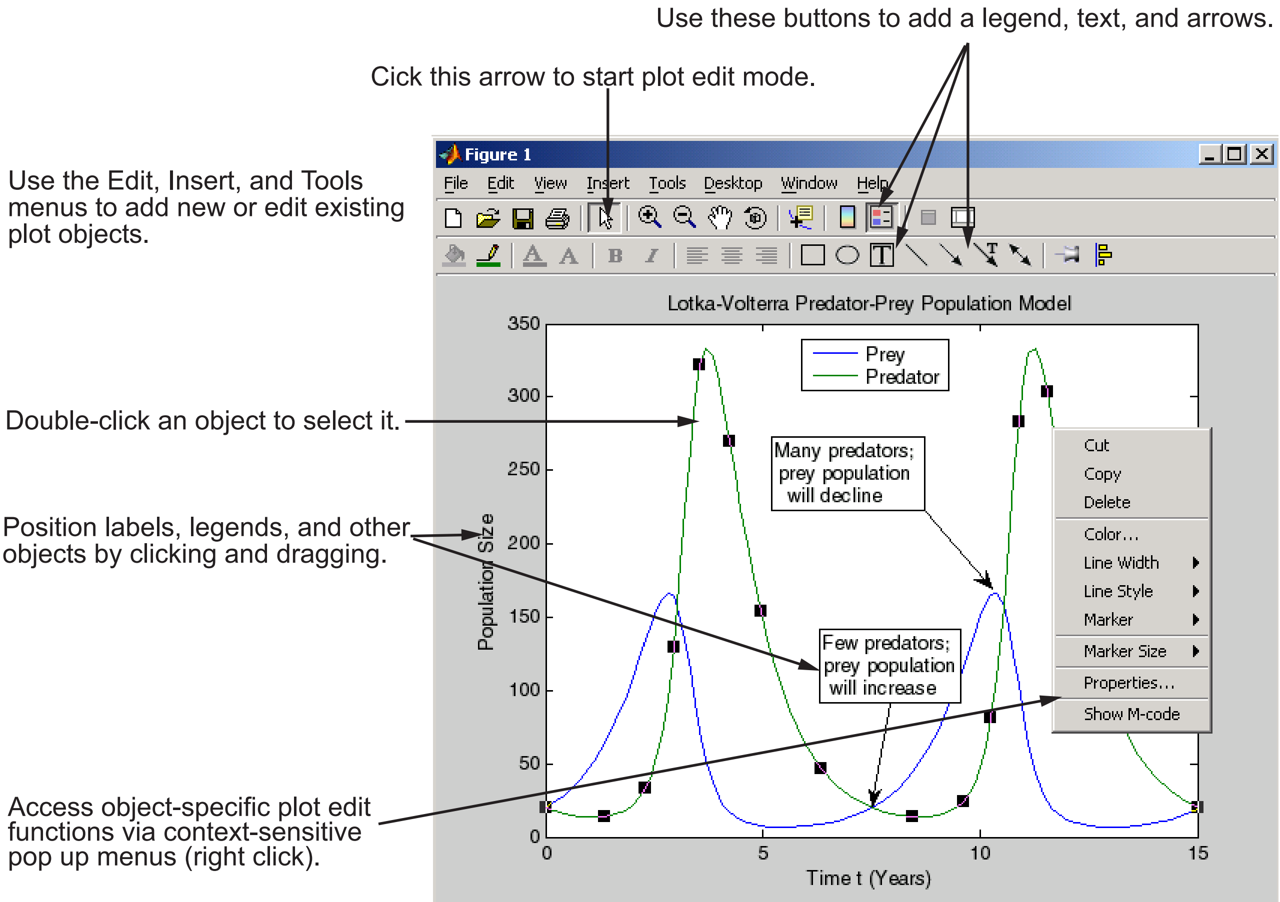
Note Plot editing mode provides an alternative way to access the properties of MATLAB graphic objects. However, you can only access a subset of object properties through this mechanism. You might need to use a combination of interactive editing and command-line editing to achieve the effect you desire.

Working in Plot Edit Mode

In this section...

Figure Windows in Plot Edit Mode

The MATLAB figure window supports a point-and-click editing mode that you can use to customize the appearance of your graph. This section describes how to enter plot edit mode and perform basic editing tasks, including selecting, cutting, copying, pasting, moving, and resizing objects and modifying other plot properties. The following figure illustrates some capabilities of plot edit mode.



Starting Plot Edit Mode

Before you can select objects in a figure by clicking on them, you must activate plot editing mode. There are several ways to activate plot edit mode:

- Choose the **Edit Plot** option on the figure window **Tools** menu.
- Click the selection button in the figure window toolbar.



- Choose an option from the **Edit** or **Insert** menu. For example, if you choose the **Axes Properties** option on the **Edit** menu, MATLAB activates plot edit mode and the axes appear selected.
- Run the `plotedit` command in the MATLAB Command Window.
- Start the plotting tools with the `plottools` command.

When a figure window is in plot edit mode, the **Edit Plot** option on the **Tools** menu is checked and the selection button in the toolbar is depressed.

Exiting Plot Edit Mode

To exit plot edit mode, click the selection button or click the **Edit Plot** option on the **Tools** menu. When plot edit mode is turned off, the selection button is no longer depressed.

Selecting Objects in a Graph

To select an object in a graph,

1. Start plot edit mode.
2. Move the cursor over the object and click it.

   Selection handles appear on the selected object.

Note When you manually select an object, its `Selected` property is set to `on`. Handles appear regardless of the setting of its `SelectionHighlight` property (which when `off` prevents handles from appearing outside of plot edit mode). Plot edit mode does not consider objects selected with `set` (as in `set(h,'Selected','on')`) to be selected, even if they have selection handles. Programmatically selected objects therefore do not respond to actions such as typing **Delete**. They can be dragged, however, because doing so selects them manually.

Selecting Multiple Objects

To select multiple objects at the same time,

1. Start plot edit mode.
2. Move the cursor over an object and **Shift**+click to select it. Repeat for each object you want to select.

You can perform actions on all the selected objects. For example, to remove a textbox annotation and an arrow annotation from a graph, select the objects and then select **Delete** or **Cut** from the **Edit** menu (**Cut** keeps a copy on the clipboard, **Delete** does not).

Deselecting Objects

To deselect an object, move the cursor off the object onto the figure window background and click the left mouse button (this deselects all selected objects and selects the one you clicked). You can also **Shift**+click on a selected object to deselect it (doing this will not deselect any other object).

Cutting, Copying, and Pasting Plot Objects

To cut an object from a graph, or copy and paste an object in a graph, perform these steps:

1. Start plot edit mode.
2. Select the object.

3 Select the **Cut**, **Copy**, or **Paste** option from the **Edit** menu or use standard shortcut keys for your platform.

Alternatively, with plot edit mode enabled, you can right-click on an object and then select an editing command from the context menu associated with the object.

Copying and Pasting Multiple Objects

When you cut or copy axes and plot objects such as lineseries or barseries from a figure and paste them there or elsewhere, the results depend on what you select and the type of container into which you paste the objects.

Copy and Paste Axes. The following semantics apply to copying and pasting axes into the same or different figure:

Select, Copy, and Paste Axes	Result of Pasting Axes
Select axes Ax1 from figure Fig1, copy and paste it into Fig2, which has no axes.	New axes Ax2 is created in Fig2. Ax2 inherits all properties of Ax1, including all children. Ax1 will be selected in Fig1; Ax2 will be selected in Fig2.
Select axes Ax1 from figure Fig1, copy and paste (from Edit Menu) into Fig2, that contains axis Ax2 which is not selected.	New axes Ax3 is created in Fig2. All children of Ax1 are copied to Ax3. All the selected objects in Fig2 are deselected, and the pasted axes Ax3 is selected. The selections in Fig1 are unchanged.
Select axes Ax1 from figure Fig1, copy and paste into Ax2 in the same or different figure.	New axis Ax3 is created having the same properties (including position) and children as Ax1; any selected objects in Ax2 are deselected, and axes Ax3 is selected. When pasting to a new figure, selections in Fig1 will be unchanged.
Select axes Ax1 from figure Fig1, copy and paste into same or different axes in Fig1.	New axes Ax2 is added to Fig1, offset from Ax1, and is the only selected object.

Note When an axis is pasted into an existing axes, the pasted axes becomes a peer of the existing axes and is offset slightly to visually indicate that the paste operation was successful.

Copy and Paste Plot Objects. The following semantics apply to copying and pasting plot objects (lines are used as examples) from one axes into the same or different figure:

Select, Copy, and Paste Objects	Result of Pasting Objects
Select and copy one or more lines from axes Ax1 and paste into selected axes Ax2 in the same or different figure.	The lines are added to Ax2; the pasted lines are the only selected objects in the destination figure.
Select and copy lines from axes Ax1 in Fig1 and paste into figure Fig2, which contains no axes or has no axes selected.	New axes Ax2 is created in Fig2 containing the lines, which are selected in it; Ax2 has default axes properties.
Select and copy lines from axes Ax1 and paste into selected axes Ax2 and Ax3.	Lines are pasted into both Ax2 and Ax3; all the pasted lines are selected.
Select and copy lines from axes Ax1 and paste into selected axes Ax1.	Nothing is pasted, as the extra content would be redundant.

Copy and Paste Plot Objects from Multiple Axes. The following semantics apply to copying and pasting plot objects (lines are used as examples) from one or more axes into the same or different figure:

Select, Copy, and Paste Objects	Result of Pasting Objects
Select and copy Line1 from axes Ax1 and Line2 from axes Ax2 and paste into axes Ax3.	Two lines are pasted into axes Ax3 and are the only selected objects there.

Select, Copy, and Paste Objects	Result of Pasting Objects
Select and copy lines from axes Ax1 and axes Ax2 and paste into figure Fig2, which contains no axes or has no axes selected.	New axes Ax3 is created in Fig2, into which all the lines are pasted; Ax3 has default axes properties.
Select and copy multiple lines from Axes Ax1 and axes Ax2 and paste into Ax1, Ax2, or some other axes.	Only those lines that did not originate in an axes are pasted into it, and the pasted lines are the only selected objects.

Copy and Paste Multiple Axes and Plot Objects. The following semantics apply to copying and pasting several axes and selected plot objects (lines are used as examples) from one or more axes into the same or different figure:

Select, Copy, and Paste Objects	Result of Pasting Objects
Select Line1 from axes Ax1, select axes Ax2, and paste into figure Fig2, which contains no axes or has no axes selected.	Ax2 and its contents is pasted as new axes Ax3; another new axis Ax4 is created into which the line is pasted
Select Line1 from axes Ax1, select axes Ax2, and paste into axes Ax3.	Line1 is pasted into axes Ax3 and axes Ax2 is pasted as new axis Ax4.
Select axes Ax1 and Line1 from Ax1, and paste into Ax1.	New axes Ax2 is created having all the properties of Ax1 but containing Line1 as its only child.
Select axes Ax1 and Line1 from Ax1 in Fig1 and paste into figure Fig2, which contains no axes or has no axes selected.	Line1 is pasted in new axes Ax2, and Ax1 and its children (including Line1) is pasted as new axis Ax3.
Select axes Ax1 and Line1 from Ax1, and paste into axes Ax2.	New axes Ax3 is created having all the properties of Ax1 but containing Line1 as its only child.

Copying and Pasting Annotation Objects

In plot edit mode you can copy and paste annotations such as textboxes, textarrows, rectangles, and ellipses, in various combinations. If any such

objects happen to be pinned to their axes (see "Pinning — Attaching to a Point in the Graph" on page 3-22), their copies are pasted unpinned. As annotation objects are children of figures, they never create new axes when you paste them.

Moving and Resizing Objects

To move or resize an object in a graph, perform these steps:

1 Start plot edit mode.

2 Select the object. Selection handles appear on the object When the cursor is over the object, it turns into crossed arrows; outside the selection it reverts to a pointer.

To move the object, drag it to the new location. You can also nudge it one pixel up, down, left, or right with the appropriate arrow key on your keyboard. If you have selected **Snap to Layout Grid** from the **Tools** menu, each keypress makes objects move to the next grid position.

To resize the object, drag a selection handle.

You can shift-click to select multiple objects and move them as a group. Arrow keys work well for this. However, when you resize one of several selected objects, only that object changes size.

Note You can move text objects, but you cannot resize them (annotation text boxes can be resized, however). You can resize axes objects, but you can only move them by dragging their edges (or via their selection handles, one at a time).

Setting Object Properties

In MATLAB graphics, every object in a graph supports a set of properties that control the graph's appearance and behavior. For example, line objects have properties that control thickness, color, and line style.

Double-clicking on an object displays the Property Editor. To edit the properties of the axes or figure, double-click on a region that does not contain other objects.

See "The Property Editor" on page 1-28 for more information.

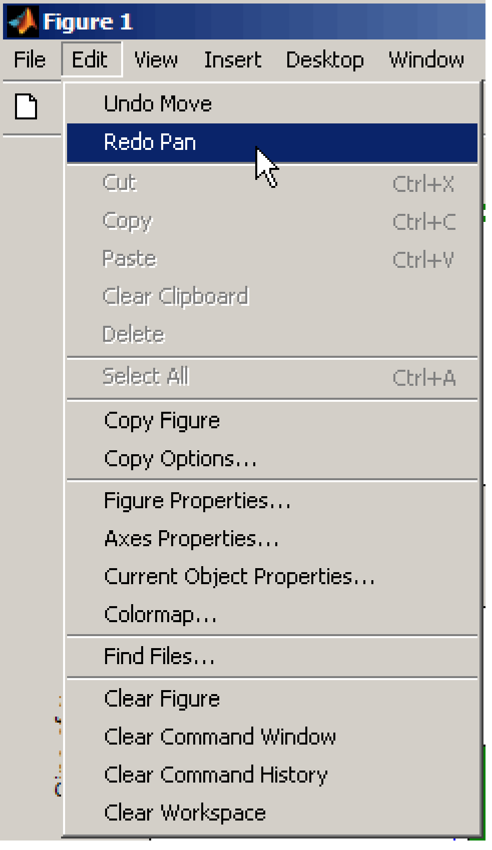
Undo/Redo — Eliminating Mistakes

The figure **Edit** menu contains two items that enable you to undo recent operation.

Undo — Remove the effect of the last operation.

Redo — Perform again the last operation that you removed by selecting **Undo**.

For example, if you create a plot, zoom in, pan the view, and then undo the pan operation, the menu looks as follows:



You could now undo the previous zoom operation or redo the pan operation you just undid.

Saving Your Work

In this section...

Saving a Graph in FIG-File Format

Note To save a figure in a format that is compatible with MATLAB versions prior to 7, refer to "Plot Objects and Backward Compatibility" on page 8-22 for more information.

The MATLAB FIG-file is a binary format to which you can save figures so that they can be opened in subsequent MATLAB sessions. The whole figure, including graphs, graph data, annotations, data tips, menus and other uicontrols, is saved. (The only exception is highlighting created by data brushing.) These files have a `.fig` filename extension.

To save a graph in a figure file,

1. Select **Save** from the figure window **File** menu or click the **Save** button on the toolbar. If this is the first time you are saving the file, the **Save As** dialog box appears.
2. Make sure that the **Save as type** is `MATLAB Figure (*.fig)` on the drop-down menu.
3. Specify the name you want to give to the figure file.
4. Click **OK**.

The graph is saved as a figure file (`.fig`), which is a binary file format used to store figures.

You can also use the `saveas` command.

Use the `hgsave` command to create backward compatible FIG-files.

If you want to save the figure in a format that can be used by another application, see "Saving to a Different Format — Exporting Figures" on page 1-70.

Opening a Figure File

To open a figure file, perform these steps:

1. Select **Open** from the **File** menu or click the **Open** button on the toolbar.
2. Select the figure file you want to open and click **OK**.

   The figure file appears in a new figure window.

You can also use the `open` command.

Saving to a Different Format — Exporting Figures

To save a figure in a format that can be used by another application, such as the standard graphics file formats TIFF or EPS, perform these steps:

1. Select **Export Setup** from the **File** menu. This dialog provides options you can specify for the output file, such as the figure size, fonts, line size and style, and output format.
2. Select **Export** from the Export Setup dialog. A standard Save As dialog appears.
3. Select the graphic format from the list of formats in the `Save as type` drop-down menu. This selects the format of the exported file and adds the standard filename extension given to files of that type.
4. Enter the name you want to give the file, less the extension.
5. Click **Save**.

Note Whenever you specify a format for saving a figure with the **Save As** menu item , that file format is used again the next time you save that figure or a new one. If you do not want to save in the previously-used format, use **Save As** and be sure to set the **Save as type** drop-down menu to the kind of file you want to write. However, saving a figure with the `saveas` function and a format does not change the **Save as type** setting in the GUI.

Copying a Figure to the Clipboard

On Microsoft® systems, you can also copy a figure to the clipboard and then paste it into another application:

1 Select **Copy Options** from the **Edit** menu. The **Copying Options** page of the **Preferences** dialog box appears.

2 Complete the fields on the **Copying Options** page and click **OK**.

3 Select **Copy Figure** from the **Edit** menu.

   The figure is copied to the Windows® clipboard. You can then paste the figure from the Windows clipboard into a file in another application.

Printing Figures

Before printing a figure,

1 Select **Print Preview** from the **File** menu to set printing options, including plot size and position, and paper size and orientation.

   The **Print Preview** dialog box opens.

2 Make changes in the dialog box. Changes you can make are arranged by tabs on the left-hand pane. If you want the printed output to match the annotated plot you see on the screen exactly,

   a On the **Layout** tab, click **Auto (Actual Size, Centered)**.

   b On the **Advanced** tab, click **Keep screen limits and ticks**.

   For information about other options for print preview, click the **Help** button in the dialog box.

To print a figure, select **Print** from the figure window **File** menu and complete the **Print** dialog box that appears.

You can also use the `print` command.

Generating an M-File to Recreate a Graph

You can generate an M-file from a graph, which you can then use to regenerate the graph. This approach is a useful way to generate M-code for work you have performed with the plotting tools. To use this option,

1. Select **Generate M-file** from the **File** menu.

   The generated code displays in the MATLAB Editor.

2. Save the M-file using **Save As** from the Editor **File** menu.

Running the Saved M-File

Most of the generated M-files require you to pass in data as arguments. The M-file assumes you are using the same data originally used to create the graph.

Comments at the beginning of the M-file state the type of data expected. For example, the following statements illustrate a case where three input vectors are required.

```
function createplot(X1, Y1, Y2)
%CREATEPLOT(X1,Y1,Y2)
%  X1:  vector of x data
%  Y1:  vector of y data
%  Y2:  vector of y data
```

See "Example — Generating M-Code to Reproduce a Graph" on page 1-53 for another example.

Data Exploration Tools

Ways to Explore Graphical Data

In this section...

Introduction

After determining what type of graph best represents your data, you can further enhance the visual display of information using the tools discussed in this section. These tools enable you to explore data interactively, eliminating the need to set the plethora of graphics properties required to achieve the same results using MATLAB commands.

Once you have achieved the desired results, you can then generate the MATLAB code necessary to reproduce the graph you created interactively. See "Example — Generating M-Code to Reproduce a Graph" on page 1-53 for more information.

Types of Tools

See the following sections for information on specific tools.

- "Data Cursor — Displaying Data Values Interactively" on page 2-5
- "Enlarging the View" on page 2-20
- "Panning — Shifting Your View of the Graph" on page 2-24
- "Rotate 3D — Interactive Rotation of 3-D Views" on page 2-26
- Camera Toolbar — Interacting with 3-D Views

You can also explore graphs visually with data brushing and linking:

- Data brushing lets you "paint" observations on a graph to select them for special treatment, such as
  - Extracting them into new variables

- Replacing them with constant or NaN values
- Deleting them

- Data linking connects graphs with the workspace variables they display, updating graphs when variables change

Brushing and linking work together across plots. When multiple graphs or subplots display the same variables, linking the graphs and brushing any of them causes the same data to also highlight on other linked graphs. The highlighting also appears on the selected rows of data when the variables are opened in the Variable Editor. For details, see "Marking Up Graphs with Data Brushing" and "Making Graphs Responsive with Data Linking" in the Data Analysis documentation.

You can perform numerical data analysis directly on graphs with curve fitting and time series tools; see

- "Regression Analysis"
- "Interactive Fitting"
- "Time Series Tools"

These and other topics are covered in the "Data Processing" section of the Data Analysis documentation. You can also use the Curve Fitting Toolbox™`cftool` if you have installed that toolbox.

Customizing Data Exploration Tools

Most of the data exploration tools place you in a *mode* in which mouse gestures and clicks control a figure's content. You can customize behavior to control what happens in a mode by setting up a *mode object* for that mode. You can find examples of how to do this in the reference pages for `datacursormode`, `pan`, `zoom` , and `rotate3d`. You can also customize the behavior of the mouse scroll wheel by writing a callback for the `WindowScrollWheelFcn` property of a figure, but your custom scroll wheel callback will be ignored while in an interactive mode.

Note Do not change figure callbacks within an interactive mode. While a mode is active (when panning, zooming, etc.), you will receive a warning if you attempt to change any of the figure's callbacks and the operation will not succeed. The one exception to this rule is the figure `WindowButtonMotionFcn` callback, which can be changed from within a mode. Therefore, if you are creating a GUI that updates a figure's callbacks, the GUI should some keep track of which interactive mode is active, if any, before attempting to do this.

Data Cursor — Displaying Data Values Interactively

In this section...

What Is a Data Cursor?

Data cursors enable you to read data directly from a graph by displaying the values of points you select on plotted lines, surfaces, images, and so on. You can place multiple datatips in a plot and move them interactively. If you save the figure, the datatips in it are saved, along with any other annotations present.

When data cursor mode is enabled, you can

- Click on any graphics object defined by data values and display the *x*, *y*, and *z* (if 3-D) values of the nearest data point.
- Interpolate the values of points between data points.
- Display multiple data tips on graphs.
- Display the data values in a cursor window that you can locate anywhere in the figure window or as a data tip (small text box) located next to the data point.
- Export data values as workspace variables.
- Print or export the graph with data tip or cursor window displayed for annotation purposes.
- Edit the data tip display function to customize what information is displayed and how it is presented
- Select a different data tip display function

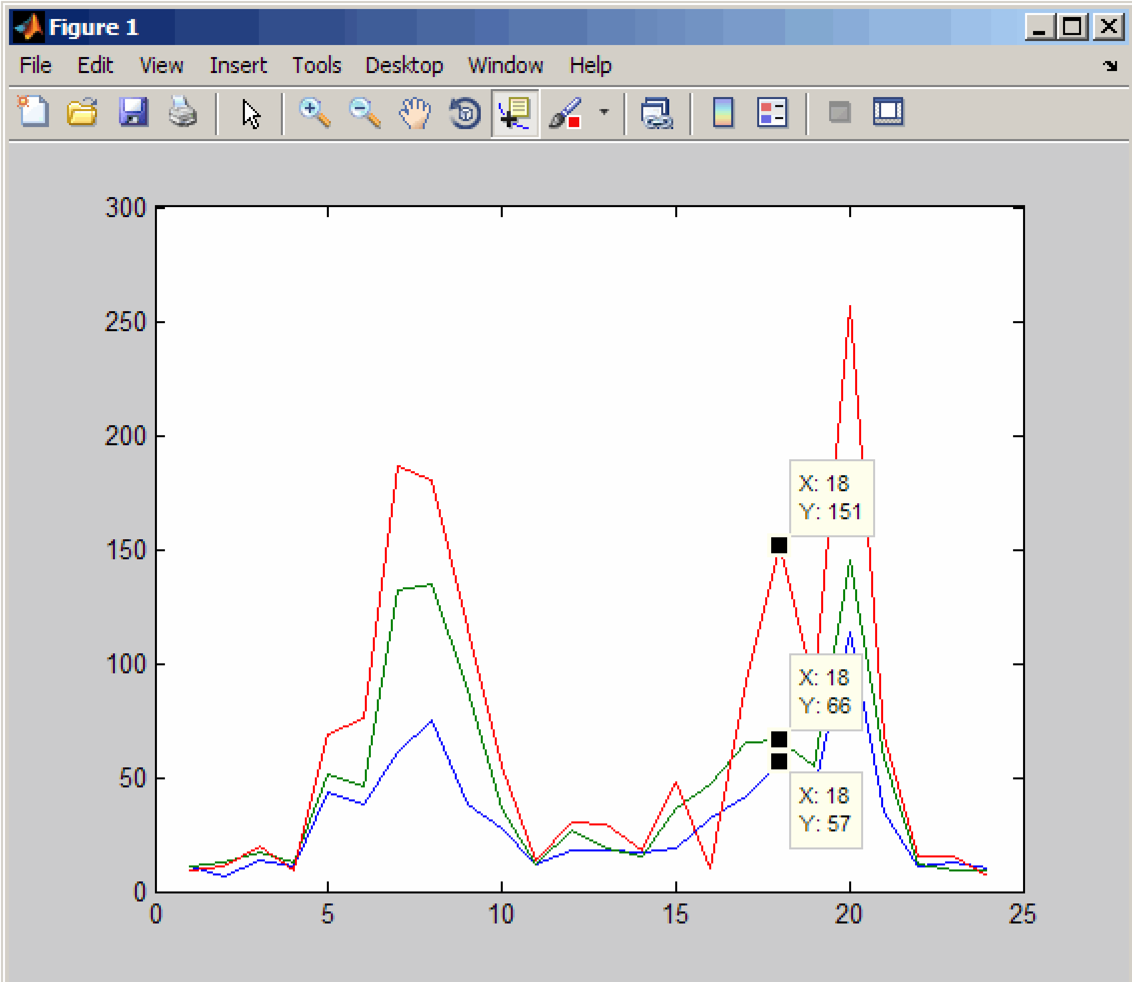
Enabling Data Cursor Mode

Select the data cursor icon in the figure toolbar  or select the **Data Cursor** item in the **Tools** menu.

Once you have enabled data cursor mode, clicking the mouse on a line or other graph object displays data values of the point clicked. Clicking elsewhere does not create or update data tips. To place additional data tips, as the picture below shows, see “Creating Multiple Data Tips” on page 2-12, below. In the picture, the black squares are located at points selected by the Data Cursor tool, and the data tips next to them display the *x* and *y* values of those points.

The illustrations below use traffic count data stored in `count.dat`:

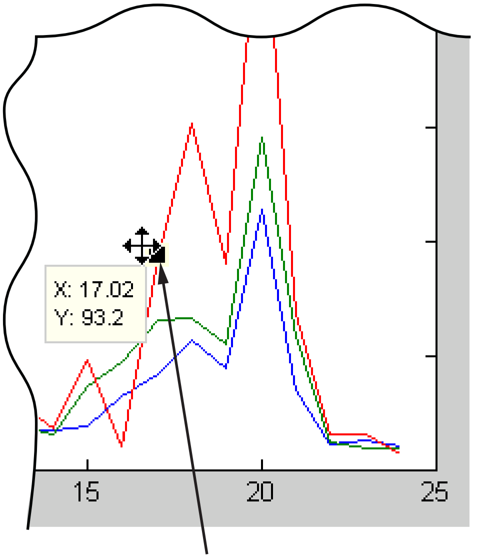
```
load count.dat
plot(count)
```



Moving the Marker

You can move the marker using the arrow keys and the mouse. The up and right arrows move the marker to data points having greater index values in the data arrays. The down and left arrow keys move the marker to data points having lesser index values. When you set **Selection Style** to **Mouse Position** using the tool's context menu, you can drag markers and position them anywhere along a line. However, you cannot drag markers between

different line or other series on a plot. The cursor changes to crossed arrows when it comes close enough to a marker for you to drag the datatip, as shown below:

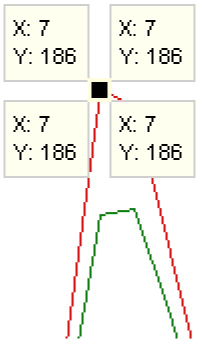


Click on the square and drag the data tip along the red line.

Positioning the Datatip Text Box

You can position the data tip text box in any one of four positions with respect to the data point: upper right (the default), upper left, lower left, and lower right.

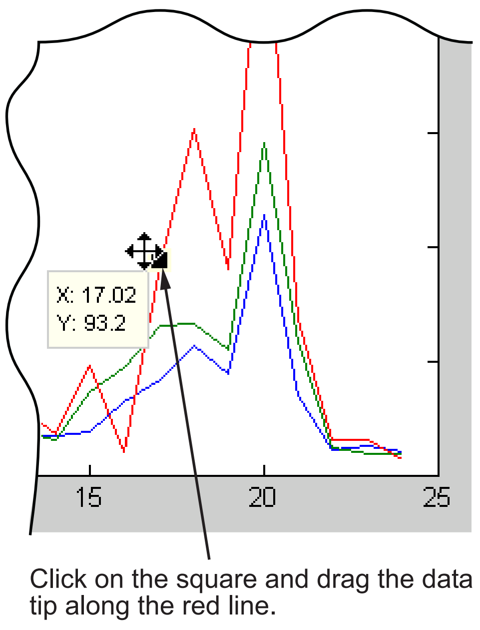
To position the datatip, press, but do not release the mouse button while over the datatip text box and drag it to one of the four positions, as shown below:



You can reposition a datatip, but not its text box, using the arrow keys as well.

Dragging the Datatip to Different Locations

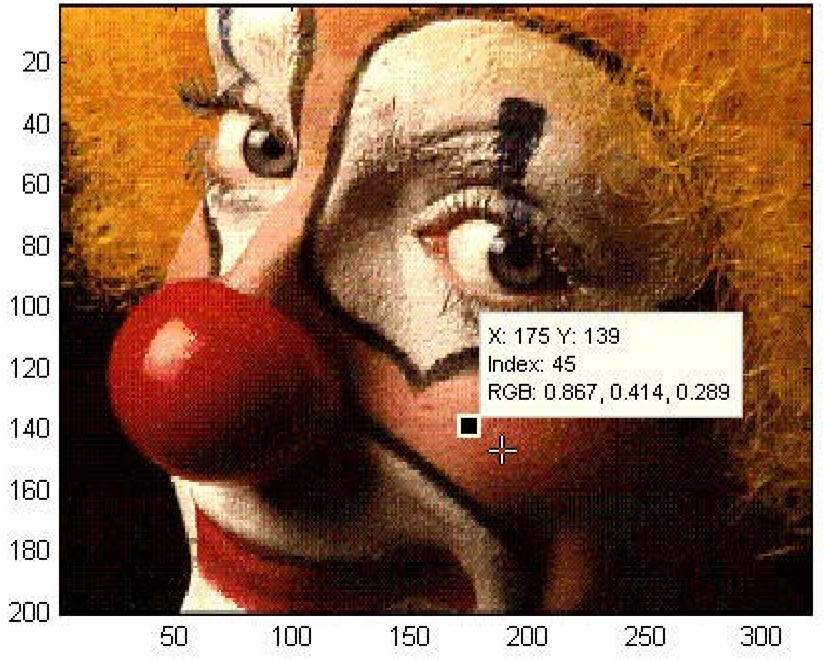
You can drag the datatip to different locations on the graph object by clicking down on the datatip and dragging the mouse. You can also use the arrow keys to move the datatip.



Note Surface plots and 3-D bar graphs can contain `NaN` values. If you drag a datatip to a location coded as `NaN`, the datatip will disappear (because its coordinates become `(NaN,NaN,NaN)`). You can continue to drag it invisibly, however, and it will reappear when it is over a non-`NaN` location. However, if you create a new datatip while the previous current one is invisible, the previous one cannot be retrieved.

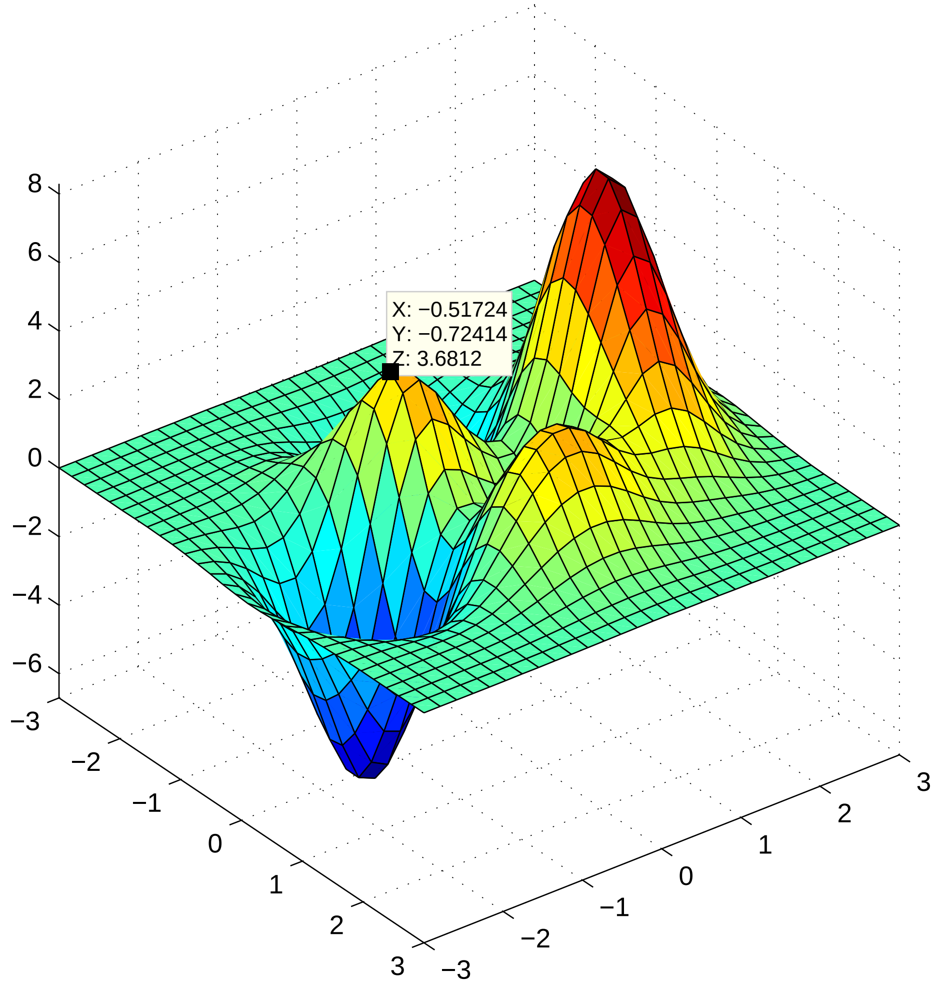
Datatips on Image Objects

Datatips on images display the *x*- and *y*-coordinates as well as the RGB values and a color index (for indexed images), as show below:



Datatips on 3-D Objects

You can use datatips to read data points on 3-D graphs as well. In 3-D views, data tips display the *x*-, *y*- and *z*-coordinates.


X: −0.51724
Y: −0.72414
Z: 3.6812
8
6
4
2
0
−2
−4
−6
−3
−2
−1
0
1
2
3
−3
−2
−1
0
1
2
3

Creating Multiple Data Tips

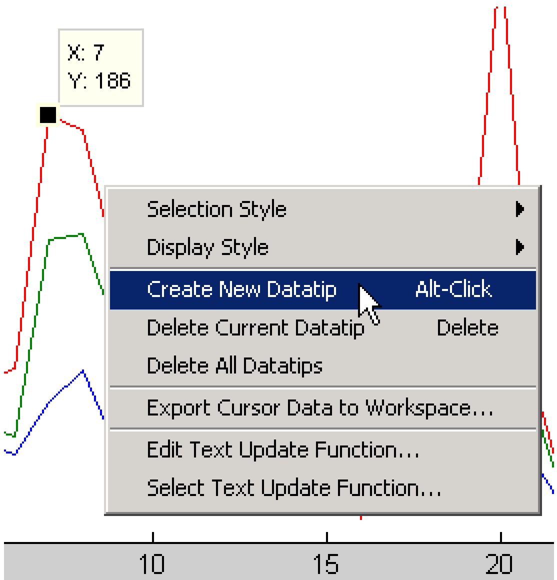
Normally, there is only one datatip displayed at one time. However, you can display multiple datatips simultaneously on a graph. This is a simple way to annotate a number of points on a graph.

Use the following procedure to create multiple datatips.

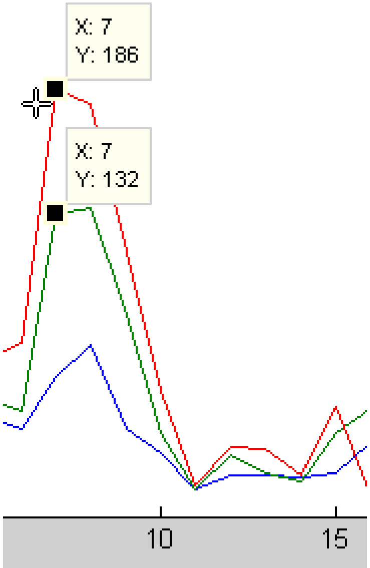
1 Enable data cursor mode from the figure toolbar. The cursor changes to a cross.

2 Click on the graph to insert a datatip.

3 Right-click to display the context menu. Select **Create New Datatip**.



4 Click on the graph to place the second datatip.



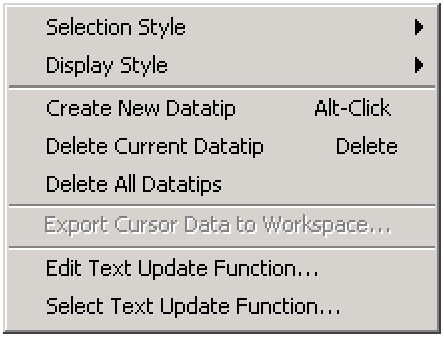
Deleting Datatips

You can remove the most recently added datatip or all datatips. When in data cursor mode, right-click to display the context menu.

- Select **Delete Current Datatip** or press the **Delete** key to remove the last datatip that you added.
- Select **Delete All Datatips** to remove all datatips.

Customizing Data Cursor Text

You can customize the text displayed by the data cursor using the `datacursormode` function. Use the last two items in the Data Cursor context menu to for this purpose:



- **Edit Text Update Function** — Opens an editor window to let you modify the function currently being used to place text in datatips
- **Select Text Update Function** — Opens an input file dialog for you to navigate to and select an M-file to use to format text in datatips you subsequently create

When you select **Edit Text Update Function** for the first time, an editor window opens with the default text update callback, which consists of the following code:

```
function output_txt = myfunction(obj,event_obj)
% Display the position of the data cursor
% obj          Currently not used (empty)
% event_obj    Handle to event object
% output_txt   Data cursor text string (string or cell array of strings).

pos = get(event_obj,'Position');
output_txt = {['X: ',num2str(pos(1),4)],...
    ['Y: ',num2str(pos(2),4)]};

% If there is a Z-coordinate in the position, display it as well
if length(pos) > 2
    output_txt{end+1} = ['Z: ',num2str(pos(3),4)];
end
```

You can modify this code to display properties of the graphics object other than position. If you want to do so, you should first save this code to an M-file

before changing it, and select that file if you want to revert to default datatip displays during the same session.

If for example you save it as `def_datatip_cb.m`, and then modify the code and save it to another M-file, you can then choose between the default behavior and customized behavior by choosing **Select Text Update Function** from the context menu and selecting one of the callbacks you saved.

See the Examples section of the `datacursormode` reference page for more information on using data cursor objects and update functions. Also see the example of customizing datatip text in "Using Data Tips to Explore Graphs" in the MATLAB Data Analysis documentation.

Display Style — Datatip or Cursor Window

By default, the data cursor displays values as a datatip (small text box located next to the data point). You can also display a single data value in a cursor window that is anchored within the figure window. You can place multiple datatips on a graph, which makes this display style useful for annotations.

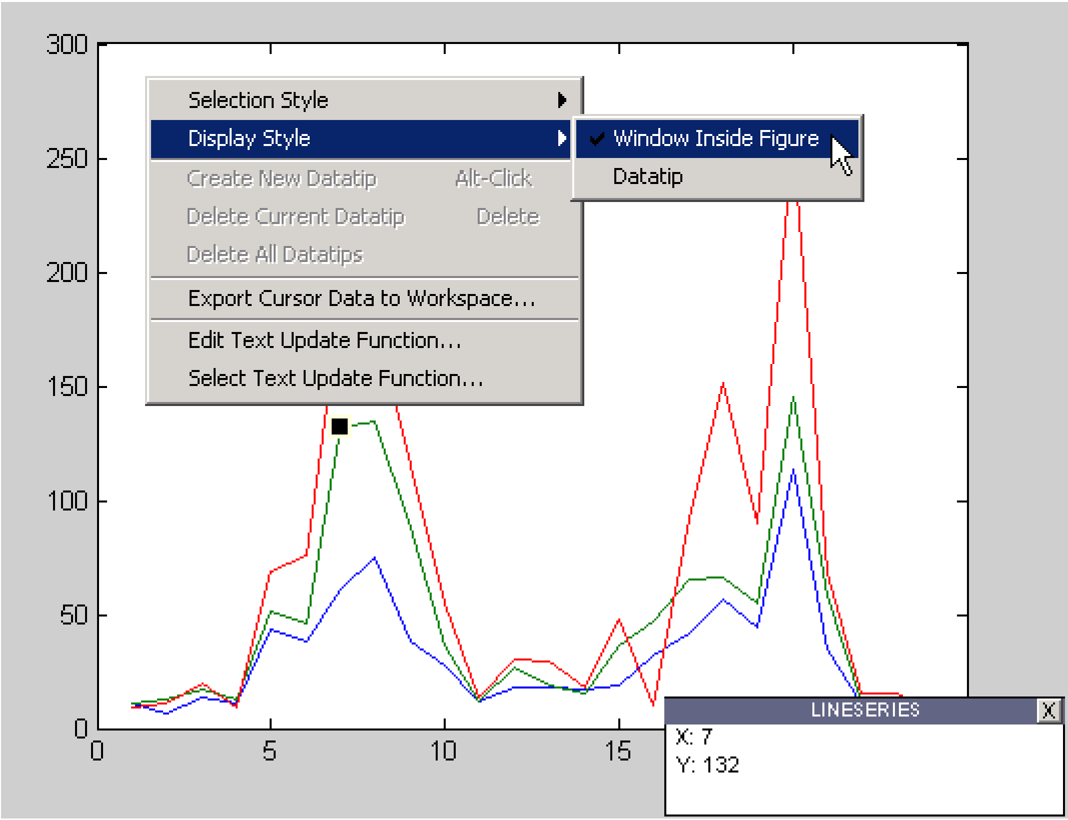
The cursor window style is particularly useful when you want to drag the data cursor to explore image and surface data; numeric information in the window updates without obscuring the any of the figure's symbology.

To use the cursor window, change the display style as follows:

1 While in data cursor mode, right-click to display the context menu.

2 Mouse over the **Display Style** item.

3 Select **Window Inside Figure**.



Note If you change the data cursor **Display Style** from **Datatip** to **Window Inside Figure** with the context menu, only the most recent data tip is displayed; all other existing data tips are removed because the window can display only one datatip at a time.

Selection Style — Select Data Points or Interpolate Points on Graph

By default, the data cursor displays the values of the data point nearest to the point you click with the mouse, and the data marker snaps to this point. The data cursor can also determine the values of points that lie in between

the data defining the graph, by linearly interpolating between the two data points closest to the location you click the mouse.

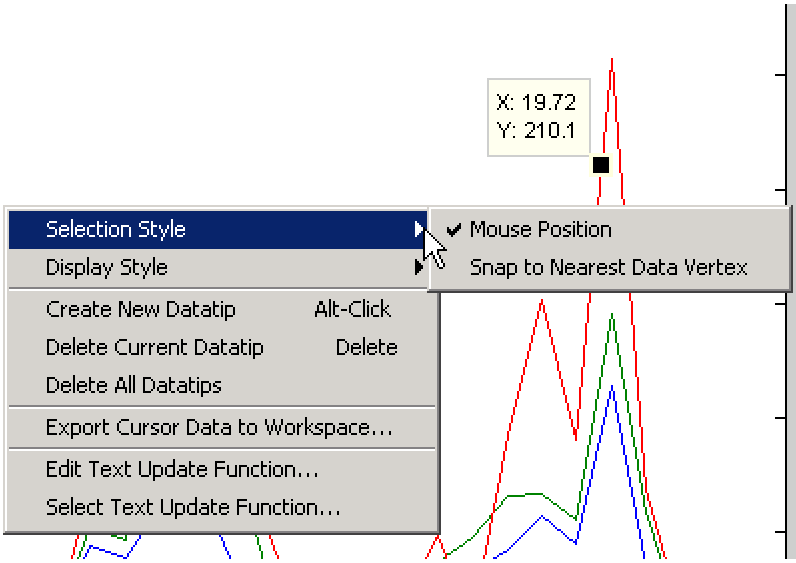
Enabling Interpolation Mode

If you want to be able to select any point along a graph and display its value, use the following procedure:

1 While in data cursor mode, right-click to display the context menu.

2 Mouse over the **Selection Style** item.

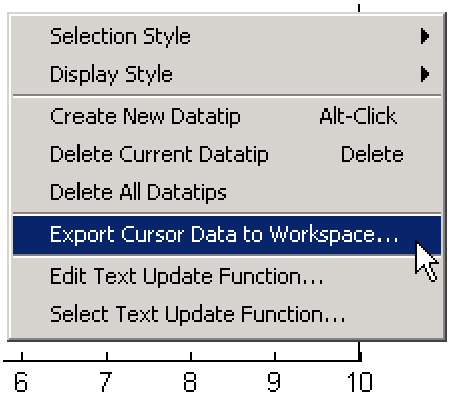
3 Select **Mouse Position**.



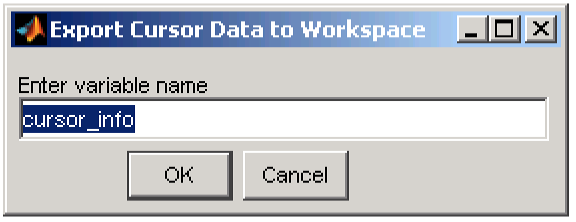
MATLAB does not honor interpolation mode when you use the arrow keys to move a datatip to a new location.

Exporting Data Value to Workspace Variable

You can export the values displayed with the data cursor to MATLAB workspace variables. To do this, display the right-click context menu while in data cursor mode and select **Export Cursor Data to Workspace**.



The Export Cursor Data to Workspace dialog then displays for you to name the workspace variable.



Clicking **OK** creates a MATLAB structure with the specified name in your base workspace, containing the following fields:

- `Target` — Handle of the graphics object containing the data point
- `Position` — *x*- and *y*- (and *z*-) coordinates of the data cursor location in axes data units

Line and lineseries objects have an additional field:

- `DataIndex` — A scalar index into the data arrays that correspond to the nearest data point. The value is the same for each array.

For example, if you saved the workspace variable as `cursor_info`, then you would access the position data by referencing the `Position` field.

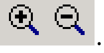
```
cursor_info.Position
ans =
    0.4189  0.1746  0
```

Enlarging the View

In this section...

Zooming in 2-D and 3-D

Zooming changes the magnification of a graph without changing the size of the figure or axes. Zooming is useful to see greater detail in a small area. As explained below, zooming behaves differently depending on whether it is applied to a 2-D or 3-D view.

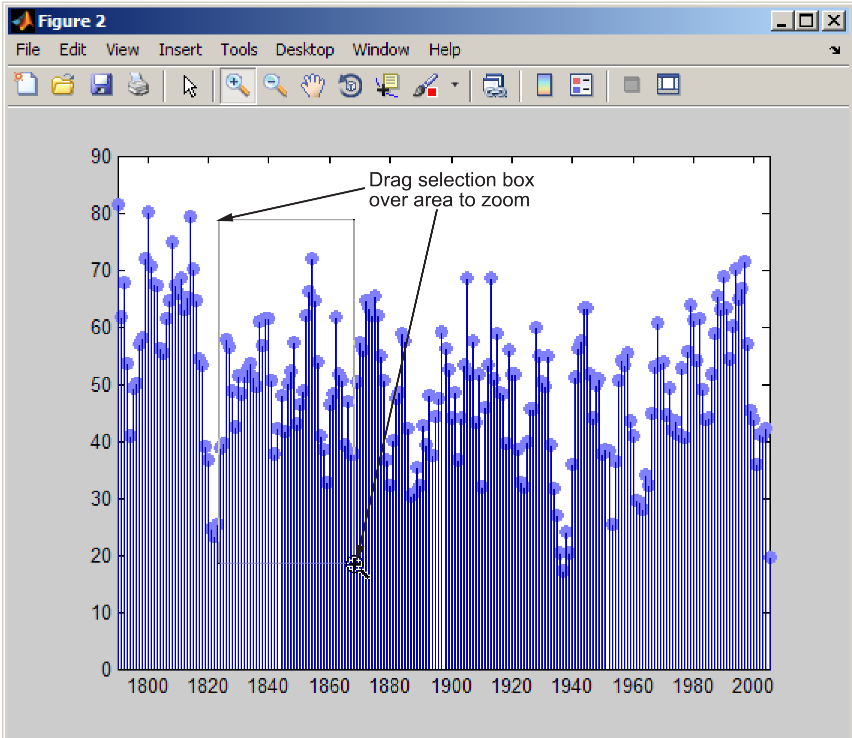
Enable zooming by clicking one of the zoom icons . Select **+** to zoom in and – to zoom out.

Tip When in zoom in mode, you can use **Shift**+click to zoom out (i.e., press and hold down the **Shift** key while clicking the mouse). You can also right-click and zoom out or restore the plot to its original view using the context menu.

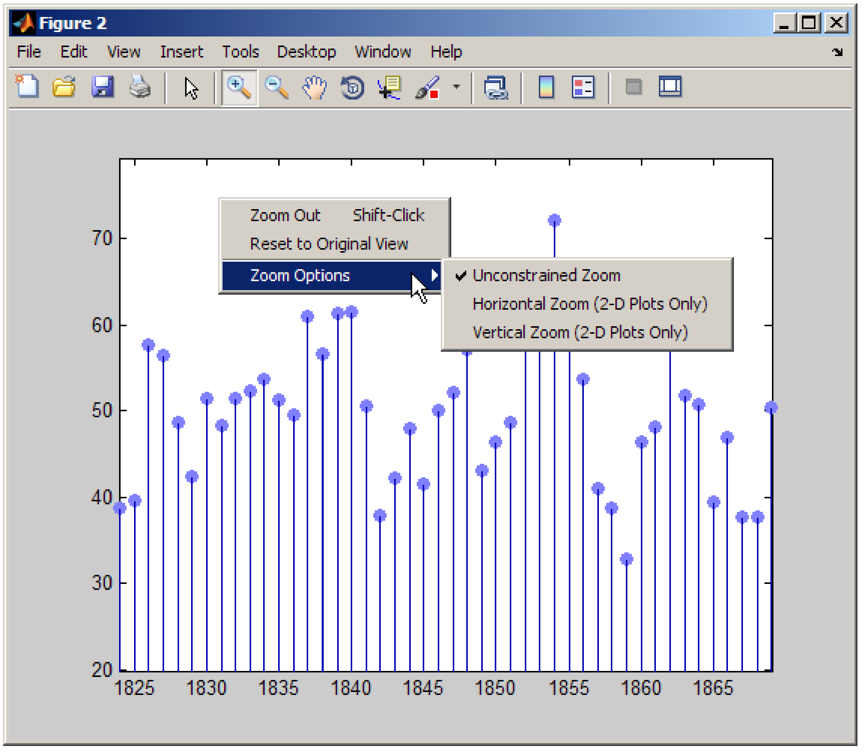
Zooming in 2-D Views

In 2-D views, click the area of the axes where you want to zoom in, or drag the cursor to draw a box around the area you want to zoom in on. The axes is redrawn, changing the limits to display the specified area.

For example, selecting the region of the following plot,



results in a rescaling of the axes to display only that region.



The above figure also shows the context menu when you right-click in Zoom mode. It enables you to

- Zoom out
- Reset to the view of the graph when it was plotted (undo one or more changes of view)
- Constrain zooming to expand only the *x*-axis (horizontal zoom)
- Constrain zooming to expand only the *x*-axis (vertical zoom)

Undoing Zoom Actions

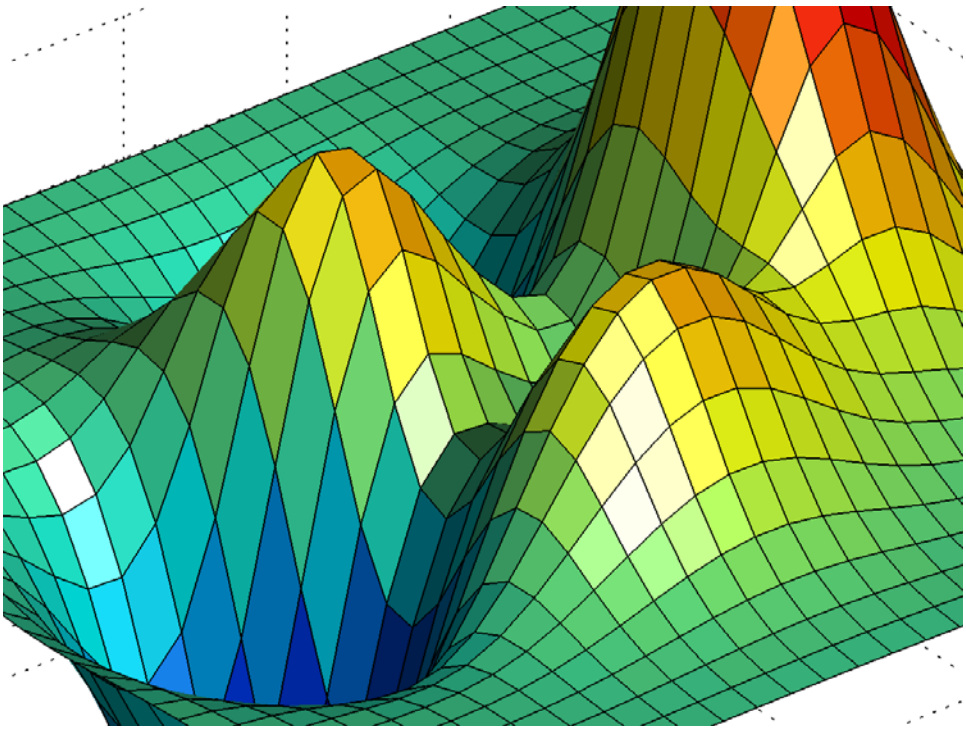
If you want to reset the graph to its original view, right-click to display the context menu and select **Reset to Original View**. You can also use the **Undo** item on the **Edit** menu to undo each operation you performed on your graph.

Zoom Constrained to Horizontal or Vertical

In 2-D views, you can constrain zoom to operate in either the horizontal or vertical direction. To do this, right-click to display the context menu while in zoom mode and select the desired constraint from the **Zoom Options** submenu, as illustrated in the previous figure. Horizontal zooming is useful for exploring time series graphs that have dense intervals. Vertical zooming can help you see minor variations in places where the `YData` range is small compared to the y-axis limits.

Zooming in 3-D Views

In 3-D views, moving the cursor up or to the right zooms in, while moving the cursor down or to the left zooms out. Both toolbar icons enable the same behavior. 3-D zooming does not change the axes limits, as in 2-D zooming. Instead it changes the view (specifically, the axes `CameraViewAngle` property) as if you were looking through a camera with a zoom lens.

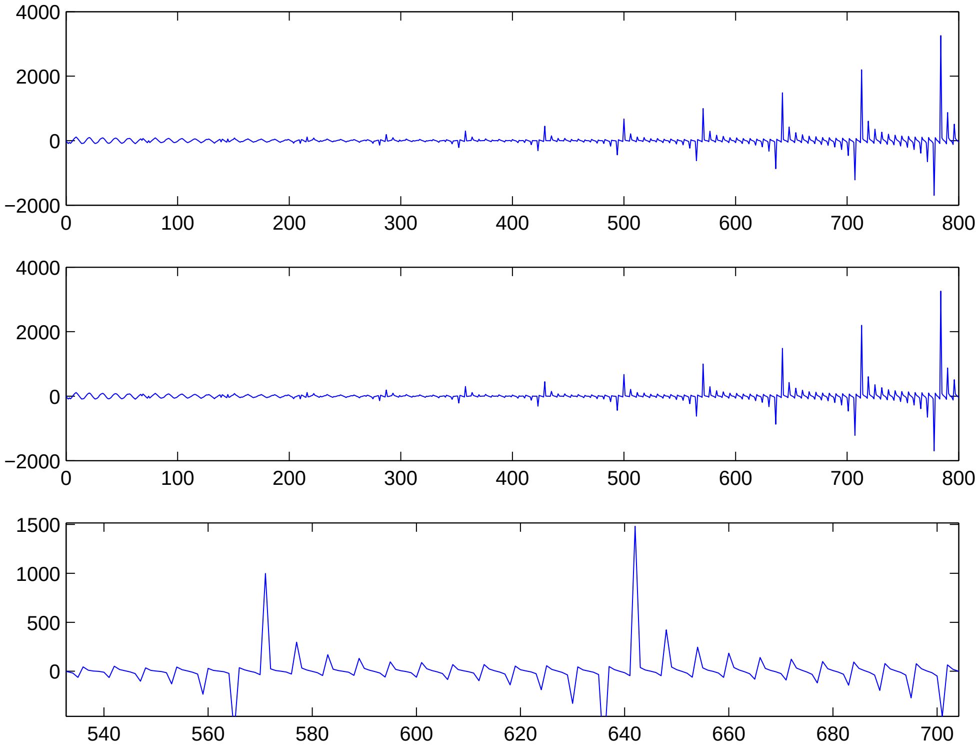


Panning — Shifting Your View of the Graph

You can move your view of a graph up and down as well as left and right with the pan tool. Panning is useful when you have zoomed in on a graph and want to translate the plot to view different portions.

Click this icon on the figure toolbar to enable panning . In pan mode you can freely move up, down, left or right, or you can constrain movement to be vertical or horizontal only by right-clicking and selecting one of the **Pan Options** from the pan tool's context menu.

You can pan across both 2-D and 3-D views. 2-D panning has the effect of changing the axis limits that you are viewing, but it does not change the actual limits of the plot. For example, suppose you have a time-series waveform that you want to zoom in on to view detail, but you also want to be able to scan the entire plot.



3-D panning moves the axes with the object, because the 3-D view is not aligned to the *x*-, *y*-, or *z*-axis. The axes limits do not change as in 2-D panning.

Rotate 3D — Interactive Rotation of 3-D Views

In this section...

Enabling 3-D Rotation

You can easily rotate graphs to any orientation with the mouse. Rotation involves the reorientation of the axes and all the graphics objects it contains. Therefore none of the data defining the graphics objects is affected by rotation; instead the orientation of the *x*-, *y*-, and *z*-axes changes with respect to the viewer.

There are three ways to enable Rotate 3D mode:

- Select **Rotate 3D** from the **Tools** menu.
- Click the Rotate 3D icon in the figure toolbar .
- Execute the `rotate3d` command.

Once the mode is enabled, you press and hold the mouse button while moving the cursor to rotate the graph.

Selecting Predefined Views

When Rotate 3D mode is enabled, you can control various rotation options from the right-click context menu.

You can rotate to predefined views on the right-click context menu:

- **Reset to Original View** — Reset to the default view (azimuth -37.5°, elevation 30°).
- **Go to X-Y View** — View graph along the *z*-axis (azimuth 0°, elevation 90°).

- **Go to X-Z View** — View graph along the *y*-axis (azimuth 0°, elevation 0°).
- **Go to Y-Z View** — View graph along the *x*-axis (azimuth 90°, elevation 0°).

Rotation Style for Complex Graphs

You can select from two rotation styles on the right-click context menu's **Rotation Options** submenu:

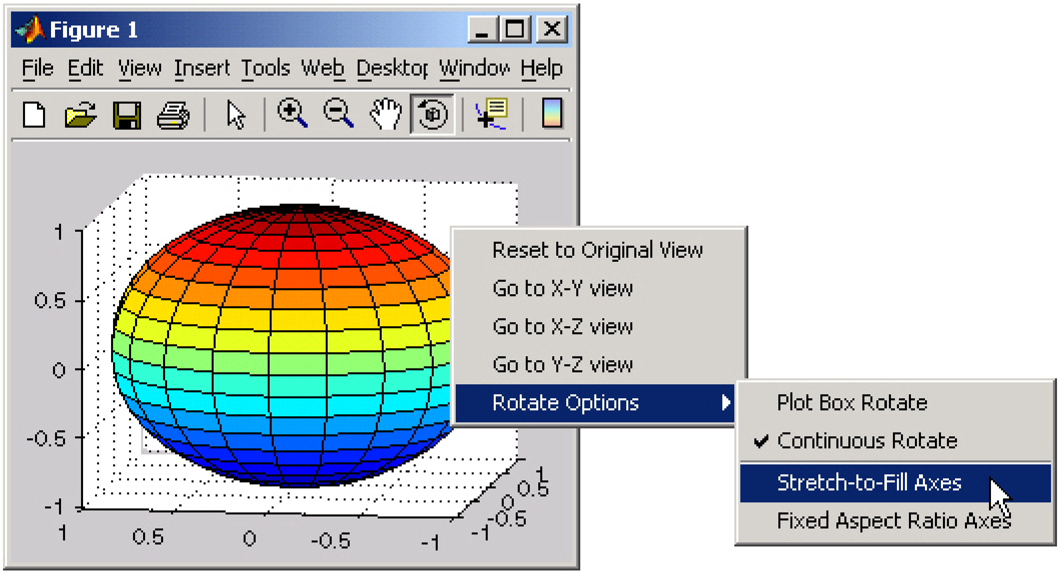
- **Plot Box Rotate** — Display only the axes bounding box for faster rotation of complex objects. Use this option if the default **Continuous Rotate** style is unacceptably slow.
- **Continuous Rotate** — Display all graphics during rotation.

Axes Behavior During Rotation

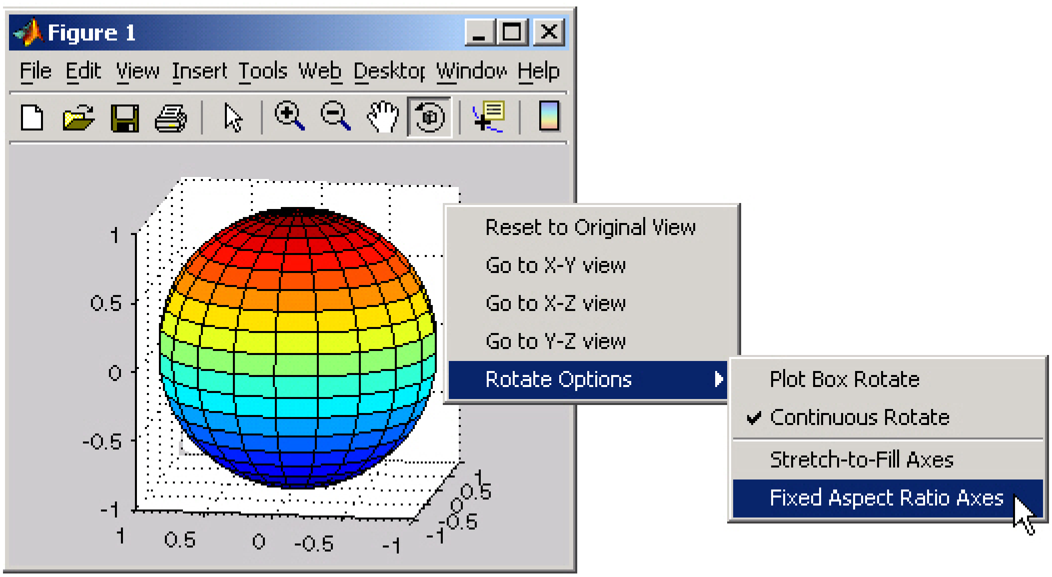
You can select two types of behavior with respect to the aspect ratio of axes during rotation:

- **Stretch-to-Fill Axes** – Default axes behavior is optimized for 2-D plots. Graphs fit the rectangular shape of the figure.
- **Fixed Aspect Ratio Axes** – Maintains a fixed shape of objects in the axes as they are rotated. Use this setting when rotating 3-D plots.

The following pictures illustrate a sphere as it is rotated with **Stretch-to-Fill Axes** selected. Notice that the sphere is not round due to the selected aspect ratio.



The next picture shows how the **Fixed Aspect Ratio Axes** option results in a sphere that maintains its proper shape as it is rotated.



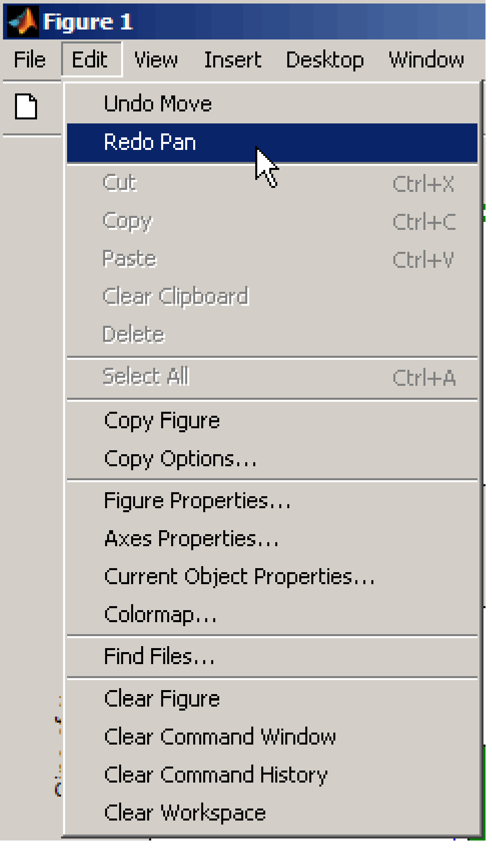
Undo/Redo — Eliminating Mistakes

The figure **Edit** menu contains two items that enable you to undo any zoom, pan, or rotate operation.

Undo — Remove the effect of the last operation.

Redo — Perform again the last operation that you removed by selecting **Undo**.

For example, if you create a plot, zoom in, pan the view, and then undo the pan operation, the menu looks as follows:



You could now undo the previous zoom operation or redo the pan operation you just undid.

3

Annotating Graphs

How to Annotate Graphs

In this section...

Graph Annotation Features

Annotating graphs with text and other explanatory material can improve the graph's ability to convey information. MATLAB graphics tools include a variety of features for annotating graphs, with which you can

- Add text, lines and arrows, rectangles, ellipses, and other annotation objects anywhere on the figure
- Anchor annotations to locations in data space
- Add a legend and colorbar
- Add axis labels and titles
- Edit the properties of graphics objects

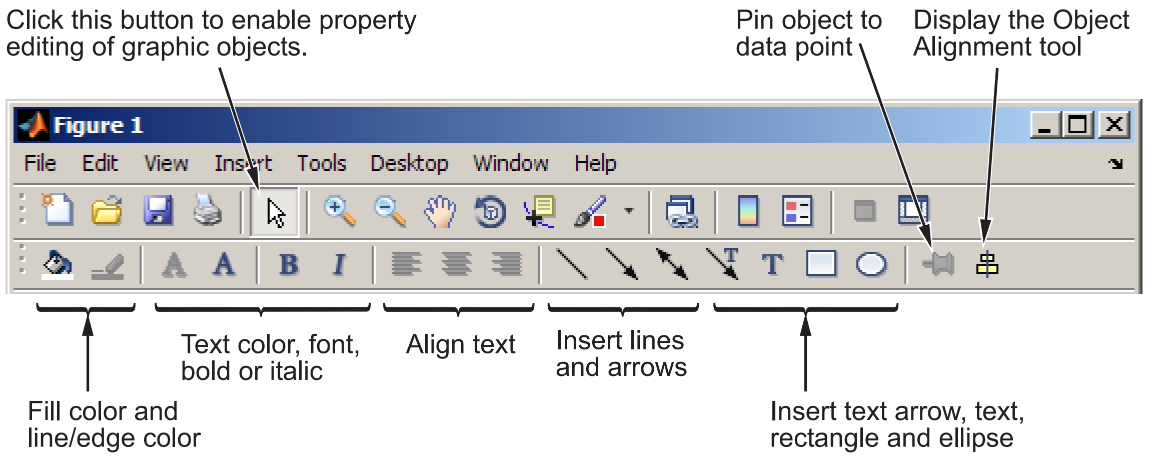
The following sections provide more information.

- "Enclosing Regions of a Graph in a Rectangle or an Ellipse" on page 3-6
- "Textbox Annotations" on page 3-8
- "Annotation Lines and Arrows" on page 3-12
- "Adding a Colorbar to a Graph" on page 3-15
- "Adding a Legend to a Graph" on page 3-19

- "Pinning — Attaching to a Point in the Graph" on page 3-22

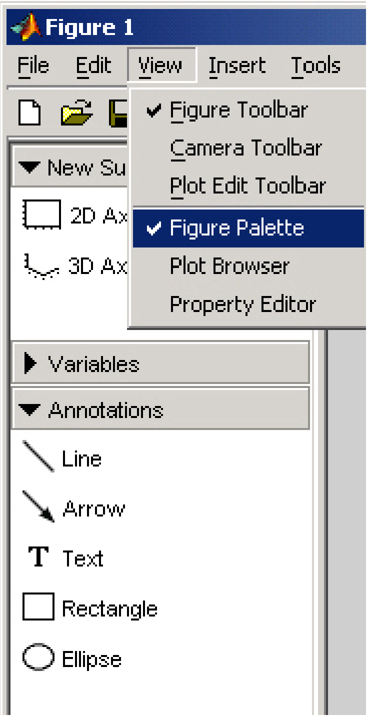
Annotation Tools on the Plot Edit Toolbar

Select **Plot Edit Toolbar** from the **View** menu to display the toolbar.



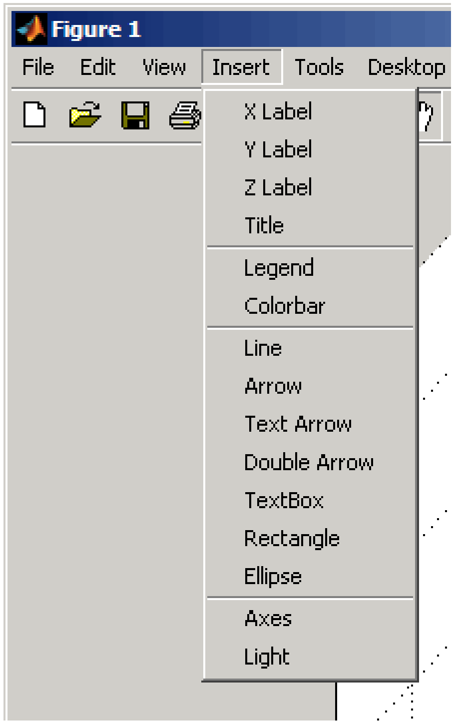
Annotation Tools on the Figure Palette

Basic annotation tools are available from the figure palette. Select **Figure Palette** from the **View** menu to display the figure palette.



Adding Annotations from the Insert Menu

Annotation features are available from the **Insert** menu.



Command Interface

You can add annotations using MATLAB commands. The following table lists the functions used to create annotations.

MATLAB Functions for Creating Annotations

Function	Purpose
`annotation`	Create annotations including lines, arrows, text arrows, double arrows, text boxes, rectangles, and ellipses
`xlabel`, `ylabel`, `zlabel`	Add a text label to the respective axis
`title`	Add a title to a graph

MATLAB Functions for Creating Annotations (Continued)

Function	Purpose
`colorbar`	Add a colorbar to a graph
`legend`	Add a legend to a graph

Removing Annotations

You can delete any annotation manually, and (if it has an explicit handle) programmatically. See "Deleting Annotations" in the MATLAB function reference documentation for details.

Enclosing Regions of a Graph in a Rectangle or an Ellipse

You can add a rectangle or an ellipse to draw attention to a specific region of a graph. While either object is selected, you can move and resize it as well as display a right-click context menu that enables you to modify behavior and appearance.

Insert the rectangle or ellipse by clicking the corresponding button in the plot edit toolbar



or by selecting **Rectangle** or **Ellipse** from the **Insert** menu. The cursor changes to a cross indicating you can click down, drag, and release the left mouse button to define the size and shape of the object.

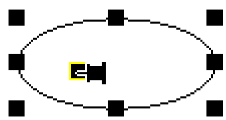
Pinning Rectangles and Ellipses

You can attach the rectangle to a particular point in the figure by pinning it to that point. There are three ways to pin the rectangle:

- Right-click the rectangle to display its context menu. Select **Pin to axes** to set a pin in the default location.

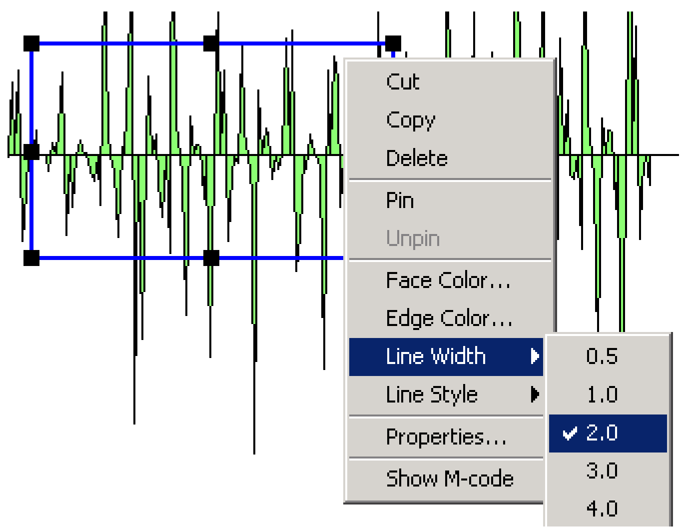
- Select the pin button in the figure toolbar (see "Pinning — Attaching to a Point in the Graph" on page 3-22).
- Select **Pin to axes** from the **Tools** menu. The cursor changes to a pin; click anywhere within the object to set a pin at that location.

By default (using the first of the options described above), pinning attaches the lower left corner of the rectangle or ellipse to its current location in the axes data units. You can move the point of attachment by clicking the corner and dragging the anchor to another point. The cursor changes to a pin while you are dragging. You cannot drag or resize a rectangle or an ellipse when it is pinned.



Modifying the Rectangle or Ellipse from the Context Menu

Right-click the rectangle or ellipse to display its context menu.



The menu contains the following options:

- **Cut**, **Copy**, **Delete** — Cut to clipboard, copy to clipboard, or delete the selected object.
- **Pin to axes** — Pin the lower left corner to the current location (you can move the point of attachment by clicking and dragging the point while in plot edit mode).
- **Unpin** — Detach the rectangle from the attachment point.
- **Face Color** — Fill color for the rectangle or ellipse
- **Edge Color** — Color of the line used to draw the rectangle or ellipse
- **Line Width** — Width of the line used to draw the rectangle or ellipse
- **Line Style** — Type of line used to draw the rectangle or ellipse
- **Properties** — Display the Property Editor with textbox properties.
- **Show M-code** — Create M-code that recreates the graph.

Setting Rectangle and Ellipse Properties

You can use the Property Editor to set rectangle and ellipse properties by selecting **Properties** from the context menu. The Property Editor displays the same properties that are described above in the context menu section.

You can click the **More Properties** button on the Property Editor to display the Property Inspector. The Property Inspector displays all properties for the selected annotation object. However, you should not change some of these properties because doing so can affect the proper functioning of the annotation object. See the following sections for descriptions of the properties you can change on the respective objects.

- Annotation Rectangle Properties
- Annotation Ellipse Properties

Textbox Annotations

A textbox is a rectangle that can contain multiline text. You can attach the textbox to any point in the figure.

Insert a textbox by clicking the textbox button in the figure toolbar **T**, then click where you want to place the text string. The default behavior for

textboxes is for them to resize to accommodate the amount of text you enter into them. You can also resize the textbox after typing or click and drag the box to a certain size when you create it (when you do this, the textbox stays that size no matter how much text you place within it).

You can also select **TextBox** from the **Insert** menu.

Selecting Textbox Objects

The selection behavior of the textbox object differs from other annotation objects.

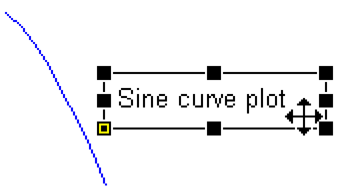
- To move a textbox, click the text once to select it.
- To edit the a textbox, double-click within the box.
- To display the Property Editor with textbox properties, right-click to display the context menu and select **Properties**.

Pinning the Textbox

You can attach the textbox to a particular point in the figure by pinning it to that point. There are three ways to pin the textbox:

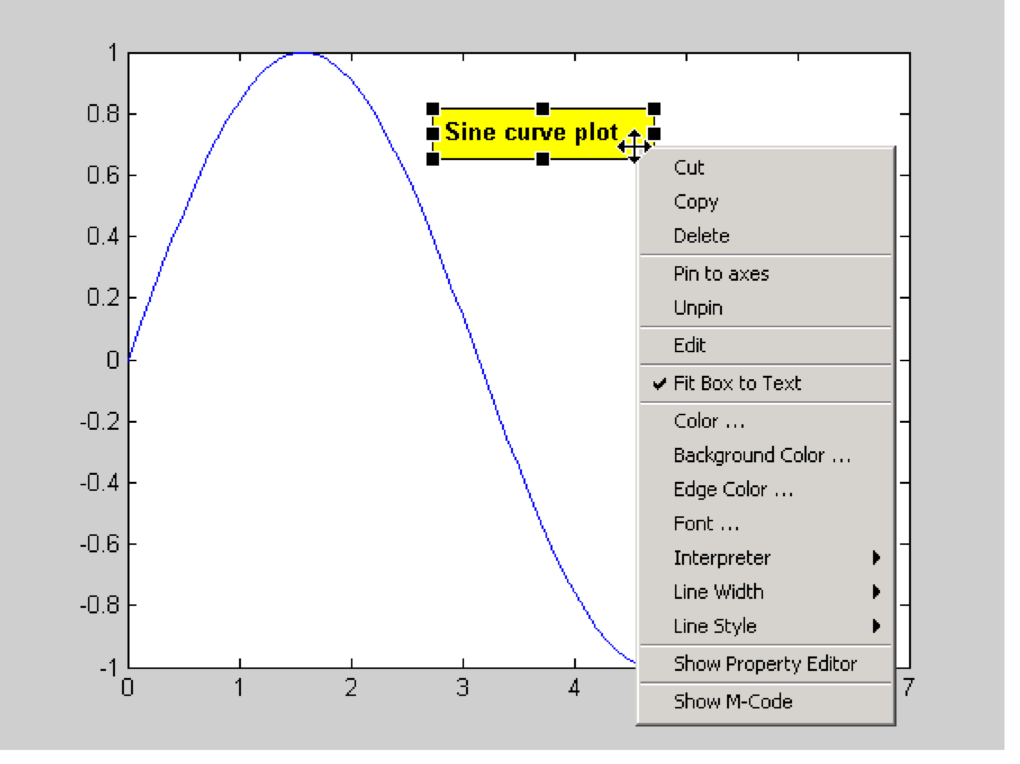
- Right-click within the textbox to display its context menu and select **Pin to Axes**.
- Select the pin button in the figure toolbar and click a handle of the textbox (See "Pinning — Attaching to a Point in the Graph" on page 3-22).
- Select **Pin Object** from the **Edit** menu.

By default, pinning attaches the lower left corner of the textbox to its location in the axes data space. You can move the point of attachment by clicking on the corner and dragging the anchor to another point, but you cannot drag the textbox when it is pinned..



Modifying the Textbox from the Context Menu

Right-clicking in a textbox displays its context menu, which enables you to perform a number of operations on the textbox. In the following picture, the textbox **Background Color** has been set to yellow and its **Font** has been set to bold using the context menu. The textbox has its default resizing behavior, as indicated by the checked item **Fit Box to Text**:



When you create a textbox without dragging it to have a specific size, **Fit Box to Text** is enabled, and the box will grow or shrink as you type or edit its text. If you drag when creating a textbox, or change its size by dragging any of its handles in plot edit mode, **Fit Box to Text** is disabled, but you can re-enable it using the context menu.

The menu contains the following options:

- **Cut**, **Copy**, **Delete** — Cut to clipboard, copy to clipboard, or delete the textbox.
- **Pin to axes** — Pin the textbox to the current location (you can move the point of attachment by clicking and dragging the textbox while in plot edit mode).
- **Unpin** — Detach the textbox from the attachment point.
- **Edit** — Enable edit mode to change the text. You can also double-click the textbox with the left mouse button to enable edit mode.
- **Fit Box to Text** — Resize textbox to accommodate text extents (or not)
- **Text Color** — Color of the text characters
- **Background Color** — Fill color of the rectangle enclosing the text
- **Edge Color** — Color of the textbox edge line (you must set **Line Style** to a value other than **none** to display edges)
- **Font** — Type of font used for the text
- **Interpreter** — Interpret characters as $\TeX$ (`latex` or `tex`) or as literal characters (`none`).
- **Line Width** — Width of the textbox edge line
- **Line Style** — Style of line used for the textbox edge
- **Properties** — Display the Property Editor with textbox properties.
- **Show M-code** — Create M-code that recreates the graph.

Setting Textbox Properties

You can use the Property Editor to set textbox properties by selecting **Show Property Editor** from the textbox context menu. It displays the same properties that are described above in the context menu section.

You can click the **More Properties** button on the Property Editor to display the Property Inspector. The Property Inspector displays all textbox properties. However, you should not change some of these properties because doing so can affect the proper functioning of the textbox.

See Textbox Properties in the reference documentation for a description of the properties you can change.

Annotation Lines and Arrows

You can add lines and three types of arrows to a graph and attach them to any point in the figure. The three types of arrows include

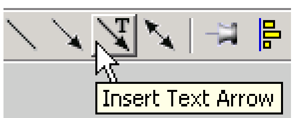
- Single-headed arrow
- Arrow with attached text box
- Double-headed arrow

Insert a line or arrow by clicking the appropriate button in the figure toolbar , then click down, drag the line or arrow to the desired point, and release the mouse. The arrowhead appears at the terminal end.

With the line or arrow selected, right-click to display the context menu, which provides access to a number of options.

Inserting a Text Arrow

A text arrow combines a textbox with an arrow. It is useful for labeling points on a graph. Add a text arrow to a graph by selecting the arrow button that has a T above the arrow. Insert the text arrow and type text in the box.

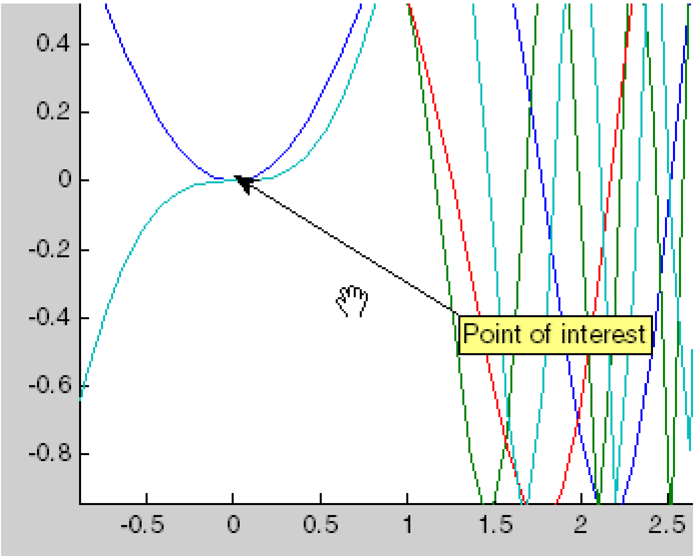


Pinning the Arrowhead End

You can attach the arrowhead end to the point of interest on the graph while letting the text box automatically reposition itself as you zoom or pan the graph.

There are three ways to pin annotations:

- Right-click the object to display its context menu and select **Pin**.
- Select the **pin** button in the plot edit toolbar (See "Pinning — Attaching to a Point in the Graph" on page 3-22).
- Select **Pin Object** from the **Edit** menu.



You can pin the arrowhead and then pan the graph to get the desired view.

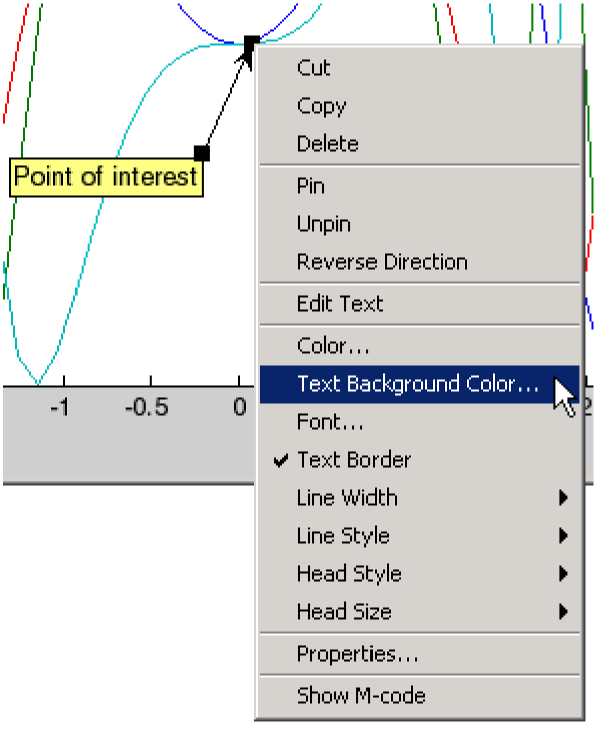
Modifying the Text Arrow from the Context Menu

Right-clicking on a text arrow displays its context menu, which enables you to perform a number of operations on the text arrow. The context menus for lines, arrows, and double arrows contain similar items:

- **Cut**, **Copy**, **Delete** — Cut to clipboard, copy to clipboard, or delete the textbox.
- **Pin to axes** — Pin the textbox to the current location (you can move the point of attachment by clicking and dragging the point while in plot edit mode).
- **Unpin** — Detach the textbox from the attachment point.
- **Reverse Direction** — Swap the arrow head and the textbox or move the arrowhead to the other end of a plain arrow.
- **Edit Text** — Enable edit mode to change the text characters.
- **Color** — Color of the text characters, textbox edge, and arrow
- **Text Background Color** — Color of the rectangle enclosing the text
- **Font** — Type of font used for the text

- **Line Width** — Width of the textbox edge line
- **Line Style** — Style of line used for the textbox edge
- **Head Style** — Type of arrowhead to use
- **Head Size** — Size of the arrowhead in points
- **Properties** — Display the Property Editor with textbox properties.
- **Show M-code** — Create M-code that recreates the graph.

For example, the following illustration shows the text border enabled and the text background color set to yellow.



Setting Line and Arrow Properties

You can use the Property Editor to set line and arrow properties by selecting **Properties** from the context menu. The Property Editor displays the same properties that are described above in the context menu section.

You can click the **More Properties** button on the Property Editor to display the Property Inspector. The Property Inspector displays all properties for the selected annotation object. However, you should not change some of these properties, because doing so can affect the proper functioning of the annotation. See the following sections in the reference documentation for descriptions of the properties you can change on the respective objects.

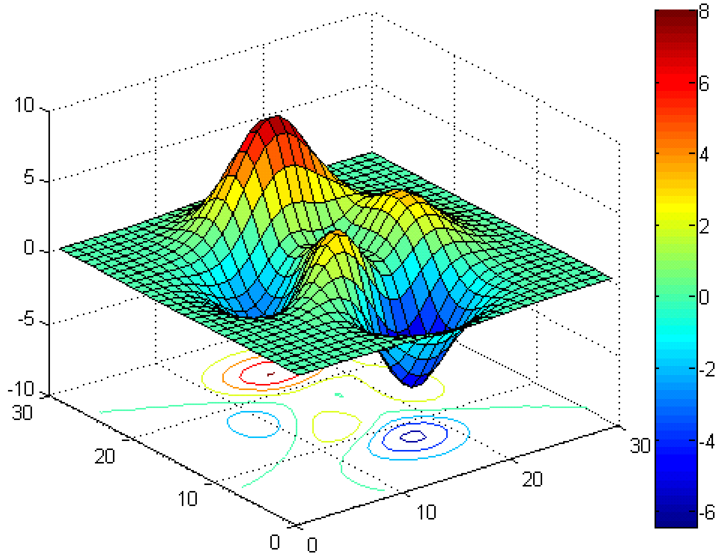
- Annotation Line Properties
- Annotation Arrow Properties
- Annotation Textarrow Properties
- Annotation Doublearrow Properties

Adding a Colorbar to a Graph

Colorbars display the current colormap and indicate the mapping from data values to colors. The following picture shows a surface plot with 2-D contour lines below. The colorbar at the right indicates how the *z*-axis data values correspond to colors in both the surface and contour graphs.

Add a colorbar by clicking the colorbar button in the toolbar  or by selecting **Colorbar** from the **Insert** menu. When plot editing is enabled, you can select and then move and resize the colorbar. The following commands will also create this plot:

```
surfc(peaks(30))
colorbar
```



You can also use the `colorbar` function to add a colorbar to a 2-D graph.

Positioning Options for Colorbars

There are a variety of ways to reposition a colorbar in the figure.

- Enable plot edit mode, then select and drag the colorbar to the desired location.
- Right-click over the colorbar to display its context menu. Mouse over **Locations** and select one of the predefined locations for the colorbar.
- Right-click over the colorbar to display its context menu and select **Properties**. This displays the Property Editor, which provides a graphical positioning device for the colorbar.

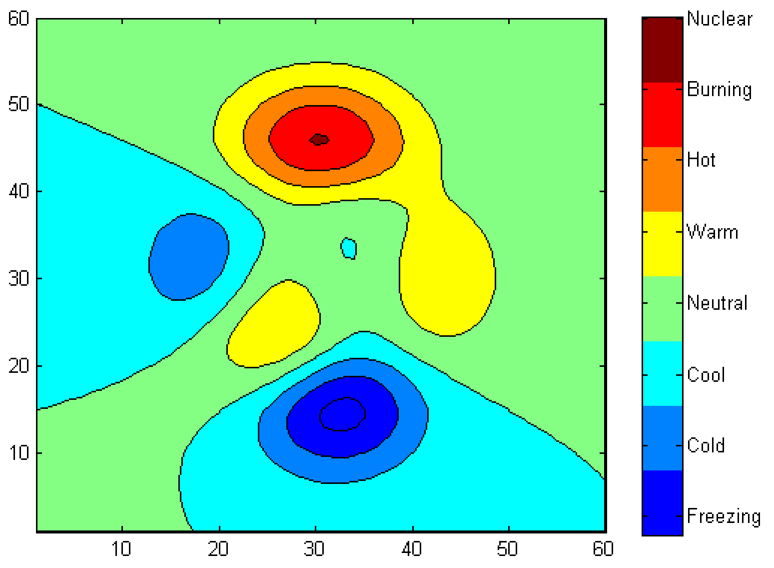
Labeling Colorbar Ticks

The default colorbar labels ticks with numeric values, which are placed at intervals specified by the colorbar's `YTick` parameter (for vertical colorbars) or its `XTick` parameter (for horizontal colorbars), within upper and lower limits specified by `CData`. You can override these limits by using the `caxis` function.

You can specify strings in place of the numeric labels on colorbars. This is useful for display of data on nominal or ordinal scales and for when you want to interpret the meaning of entries in the colormap for the viewer. To substitute strings for numbers along a colorbar, you define a label for each tick location. You do this by specifying a cell array of strings for `YTicklabel` (vertical colorbars) or `XTickLabel` (horizontal colorbars), as the examples below show.

Example 1: Default Vertical Colorbar with YTickLabels

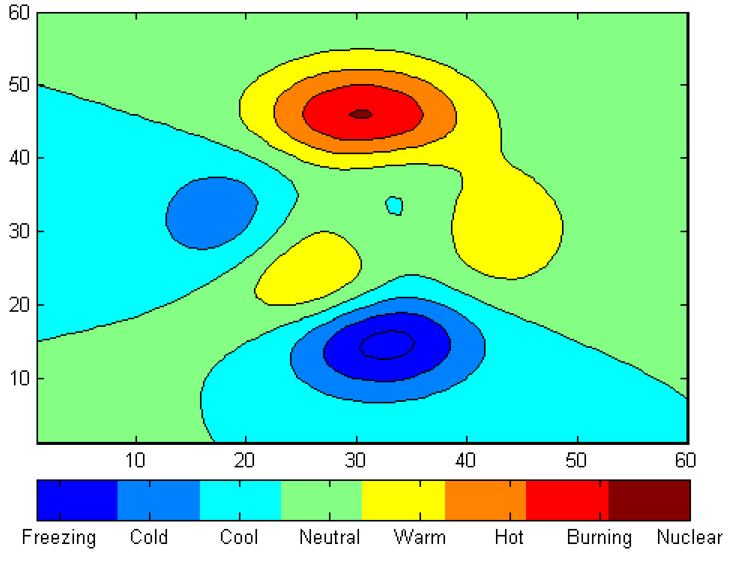
```
contourf(peaks(60));
colormap(jet(8));
hcb = colorbar('YTickLabel',...
{'Freezing','Cold','Cool','Neutral',...
'Warm','Hot','Burning','Nuclear'});
set(hcb,'YTickMode','manual')
```



Example 2: Horizontal Colorbar with XTickLabels

```
figure
contourf(peaks(60));
```

```
colormap(jet(8));
hcb = colorbar('Location','SouthOutside','XTickLabel',...
{'Freezing','Cold','Cool','Neutral',...
'Warm','Hot','Burning','Nuclear'});
set(hcb,'XTickMode','manual')
```



In these examples, the number of colors and the number of labels were set to be the same (8). This is typical for nominal (categorical) data, but not necessary if you do not object to having a range of colors associated with each label.

If ticks change, for instance if `YTick` (`XTick`) values change or the plot is rescaled while `YTickMode` (`XTickMode`) or `YTickLabelMode` (`XTickLabelMode`) is `auto`, too few or too many colorbar labels may be displayed, and can sit next to colors they do not represent. When there are fewer labels than ticks, the labels will cycle, with the lowest one following the highest one, etc., to give each tick a label. This is probably not what you want, so you need to reset `YTick` (`XTick`) values in such cases. Finding the correct values can take some experimentation. The `set` functions in the above examples prevent MATLAB from changing the number of ticks when you resize figures.

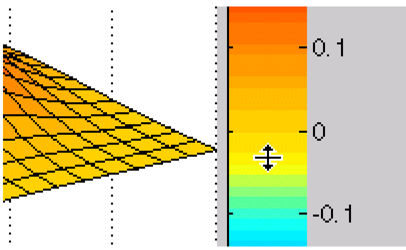
Selecting a Different Colormap

If you change the figure colormap, the colorbar updates automatically. Use one of the following methods to change the colormap.

- Right-click over the colorbar to display its context menu. Mouse over **Standard Colormaps** and select from the displayed list.
- Right-click over the colorbar to display its context menu and select **Properties**. Click the figure background to load the figure properties into the Property Editor. Select the colormap from the pull-down list.
- Use the `colormap` function.

Modifying the Colormap

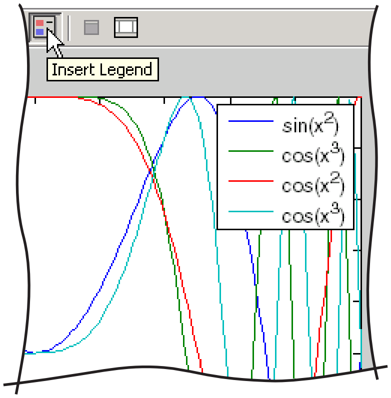
You can use a colorbar to modify the current colormap. To do this, select **Interactive Colormap Shift** from the right-click context menu. In this mode, you can left-click down on any color in the colorbar and, by dragging the mouse, shift the color-to-data mapping.

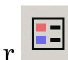


To perform more sophisticated operations on the colormap, open the colormap editor by selecting **Open Colormap Editor** from the colorbar's context menu. See the `colormapeditor` reference page for more information.

Adding a Legend to a Graph

Legends provide a key to the various data plotted on a graph. The following picture shows the legend for a graph of four functions of a variable plotted with lines of different colors. A graph can have only one legend, which applies to and will symbolize all data series contained by an axes, according to their form (e.g., lines, bars, pies, etc.). You can assign an appropriate string to each line in the legend.



Add a legend by clicking the legend button in the toolbar  or by selecting **Legend** from the **Insert** menu. When plot editing is enabled, you can select and then move and resize the legend.

You can also use the `legend` function to add a legend to a graph, which gives additional controls over appearance. You must use this command in order to display a legend with more than 50 entries, as the legend toolbar button is limited to displaying legends for 50 elements.

Specifying the Text

By default, the legend labels each plotted object (line, surface, etc.) with the strings `data1`, `data2`, etc. You can change this text by double-clicking on the text to enable edit mode. In edit mode, you can retype the text string.

You can use TeX characters in the text strings to produce symbols. You can disable interpretation of characters as TeX sequences by selecting **none** from the **Interpreter** submenu of the legend's right-click context menu.

See the Table of TEX symbols in the Text Properties reference documentation for more information.

Positioning the Legend

There are a number of ways to position the legend.

- Enable plot edit mode, select the legend, and drag it to the desired location.
- Right-click the legend to display its context menu, mouse over **Location,** and select one of the predefined locations from the submenu.
- Right-click the legend to display its context menu and select **Properties** to display the Property Editor, which provides a graphical device for positioning the legend.

You can also select a vertical or horizontal orientation for the legend. Use the **Orientation** item in the context menu to make this selection.

Changing the Appearance of the Legend

You can specify the following legend characteristics from the context menu:

- **Color** — Set the background color of the legend. In addition, you can specify the `Color` property as `'none'` to make the legend background be transparent.
- **EdgeColor** — Set the color of the line enclosing the legend box.
- **TextColor** — Set the color of the legend text.

  You can use a colorspec or an RGB color triplet to set the above three properties.
- **LineWidth** — Set the width of the edge line.
- **Font** — Set the font, font style, and font size of the text used in the legend.
- **Interpreter** — Set the text `Interpreter` property to use $\TeX$ or plain text.
- **Orientation** — Orient the legend entries side by side (horizontal) or on top of each other.
- **Properties** — Display the Property Editor with legend properties.
- **Show M-code** — Generates M-code for recreating the legend.

Controlling the Appearance of Grouped Objects on a Legend

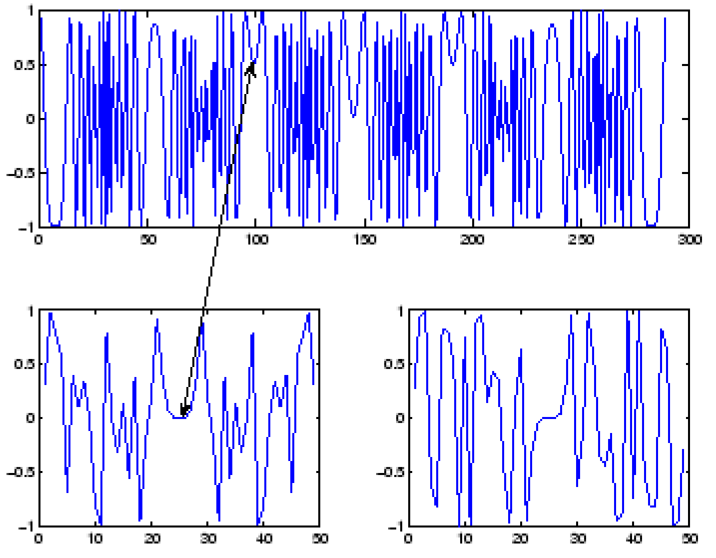
When you create a legend for groups of graphic objects such as lineseries, barseries, or stemseries, the default legend will show an individual legend entry for each of the graphics objects. Sometimes you might want only certain objects to appear in a legend, to show one legend entry for the entire series,

or to show the individual children of a series (however, not all series have children; you can use the Handle Graphics Property Browser to determine this). You can control how groups appear in the legend by setting values for their `Annotation` property via M-code. For information on how to customize legends in this manner, see “Controlling Legends” on page 8-100.

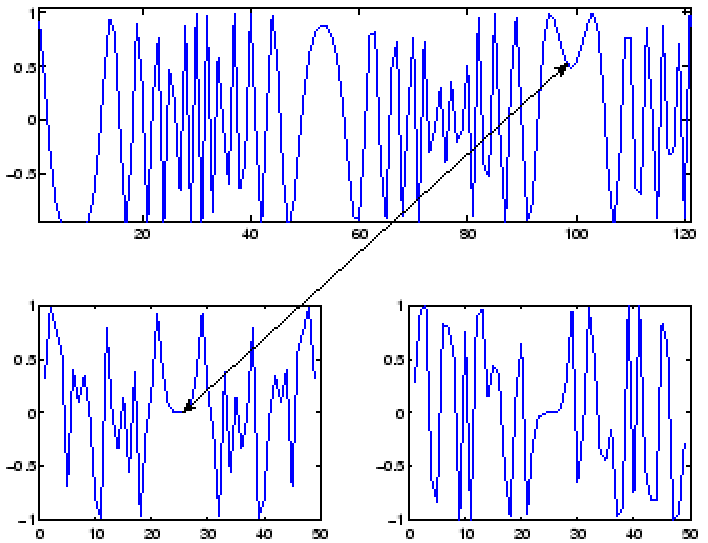
You can view the values of an object’s `Annotation` property in the Property Inspector, but you can not set them there; you need to use M-code.

Pinning — Attaching to a Point in the Graph

Pinning is the attachment of an object to a particular point in the figure. Pinning enables you to pan or resize the figure while keeping annotations associated with the same point. For example, the following picture shows regions in two different graphs associated by pinning both ends of a double arrow.



If you perform a horizontal zoom on the top axes (select **Horizontal Zoom** from the **Options** submenu of the **Tools** menu) and then pan the graph to show the first 120 seconds of the data, the double arrow continues to point to the same locations on the graph.



Pinning Objects

To pin an object, first enable pinning mode by clicking the **pin object** button  in the plot edit toolbar or selecting **Pin Object** from the **Edit** menu. Then click the point you want to pin.

To unpin an object, right-click to display the context menu and select **Unpin**.

You can pin annotation lines, arrows, rectangles, ellipses, and text boxes.

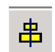
When this mode is enabled, axes, rectangle, arrows, and lines automatically align their upper left corners to the grid. As you move or resize one of these objects, the size or position snaps to the next grid location.

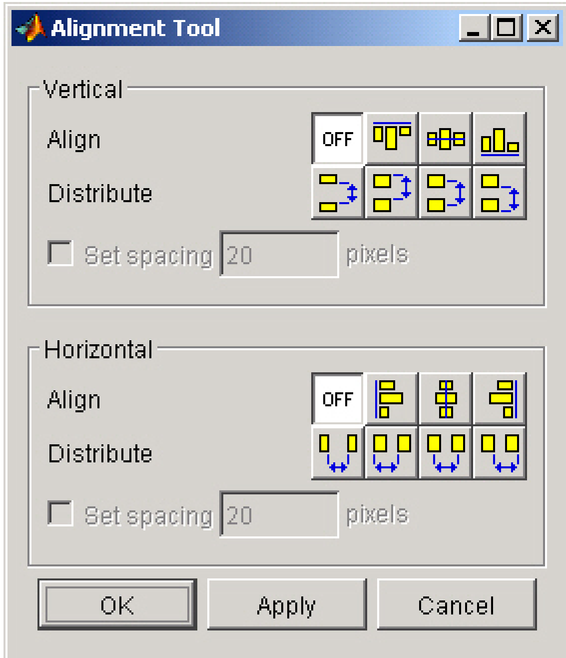
Alignment Tool — Aligning and Distributing Objects

In this section...

Alignment Tool Functionality

The Alignment Tool enables you to position objects with respect to each other and to adjust the spacing between selected objects. The specified align/distribute operations apply to all components that are selected when you click the **Apply** or **OK** buttons.

Display the Alignment Tool by clicking the Align/Distribute button  or by selecting **Align Distribute Tool** from the **Tools** menu.



The Alignment Tool provides two types of positioning operations:

- **Align** — Align all selected objects to a single reference line.
- **Distribute** — Space all selected objects uniformly with respect to each other.

You can align and distribute objects in the vertical and horizontal directions. The following sections provide more information.

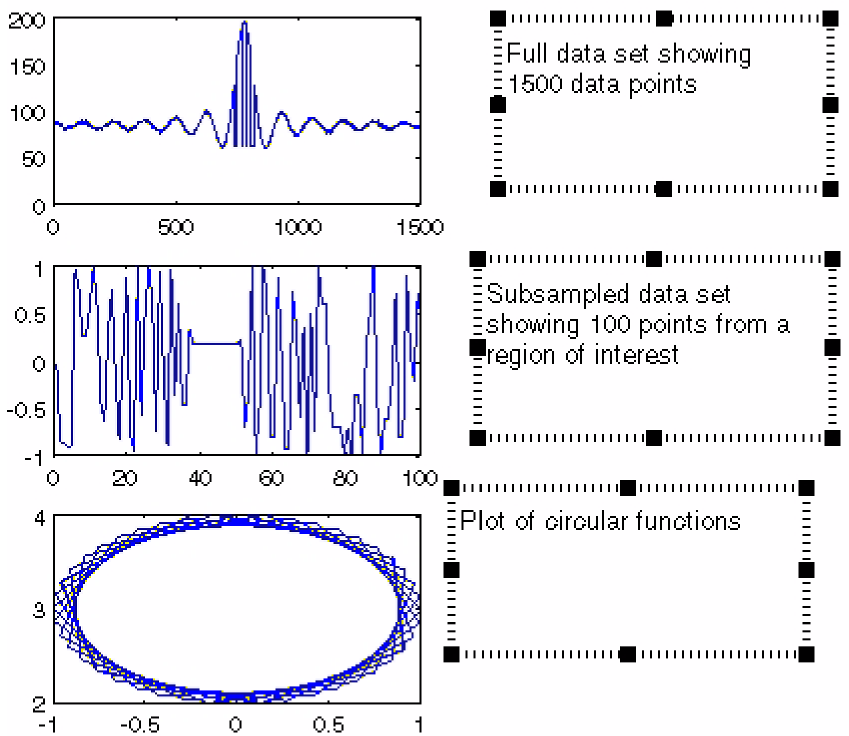
- "Example — Vertical Distribute, Horizontal Align" on page 3-25
- "Align/Distribute Menu Options" on page 3-28
- "Snap to Grid — Aligning Objects on a Grid" on page 3-30

Example — Vertical Distribute, Horizontal Align

This example illustrates how to align three textboxes with three corresponding axes. In this example, the text boxes were just plunked down close to the

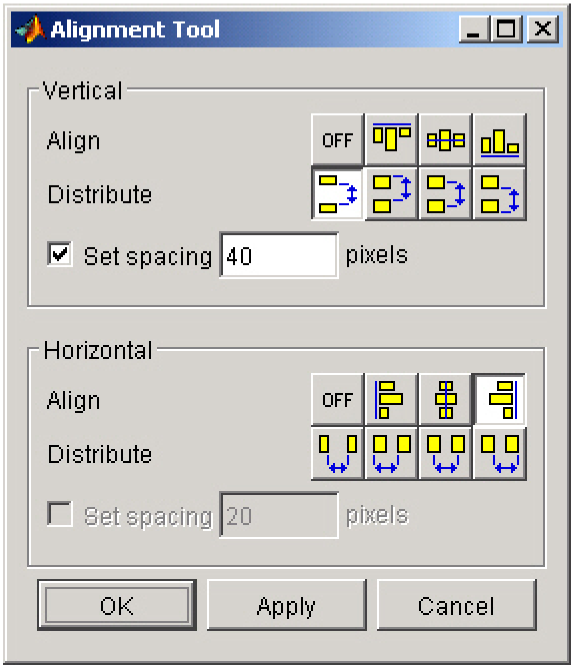
desired position and then right aligned and distributed to have 40 pixels between them.

The following picture shows the initial layout.

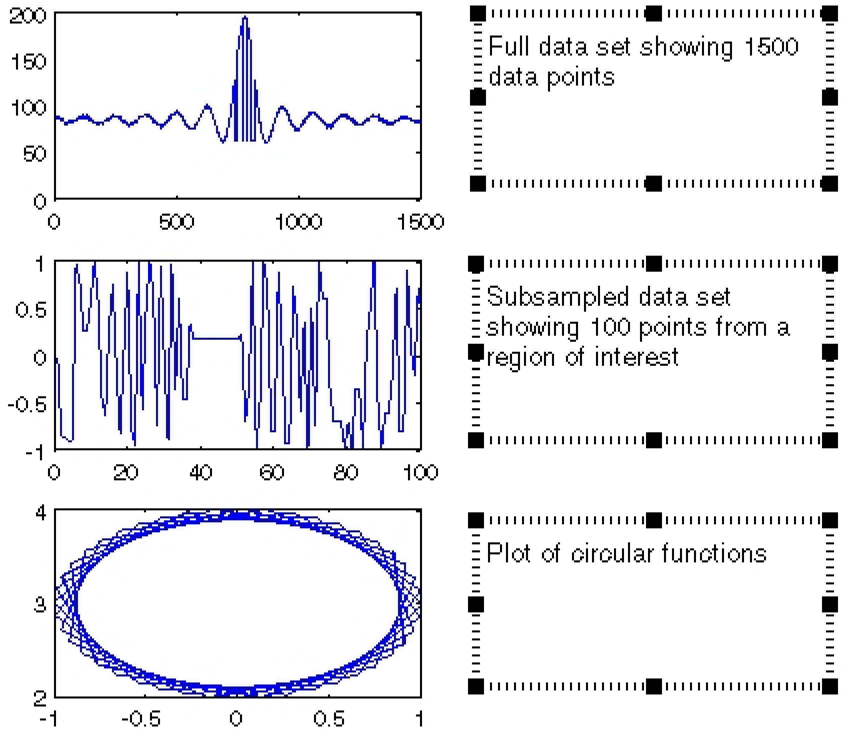


Use **Shift**+click to select all three textboxes and then configure the Alignment Tool as shown in the following picture.

- Set vertical distribution to 40 pixels.
- Set horizontal alignment to right-aligned.
- Click **Apply.**

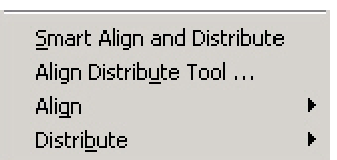

Alignment Tool
Vertical
Align
OFF
Distribute
Set spacing
40
pixels
Horizontal
Align
OFF
Distribute
Set spacing
20
pixels
OK
Apply
Cancel

The following picture shows the result.



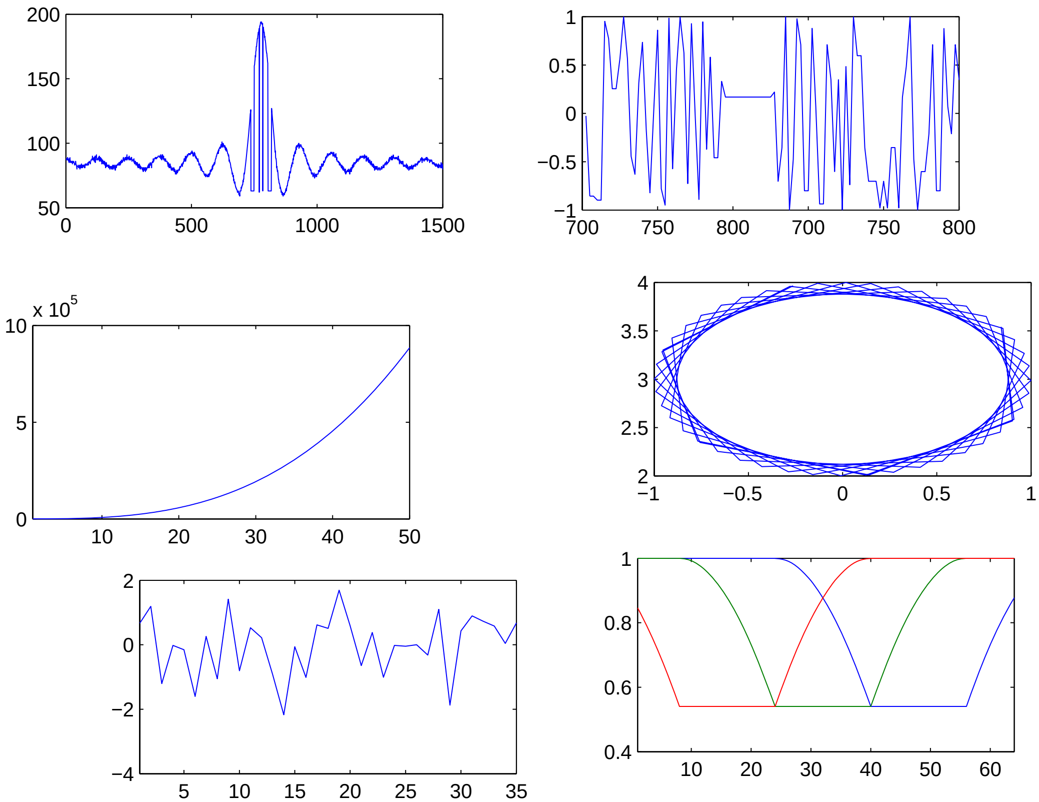
Align/Distribute Menu Options

The **Tools** menu contains the alignment and distribution options that are available via the Alignment Tool.



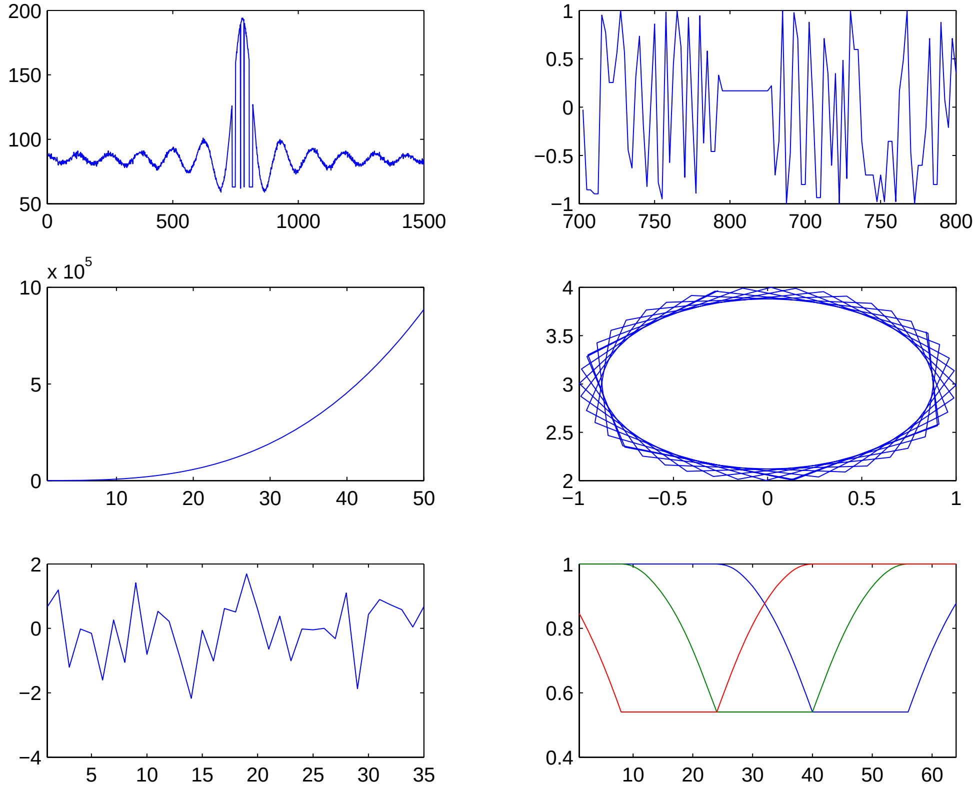
The **Smart Align and Distribute** option aligns objects into rows and columns with equal spacing between each object. It is useful when you have a number of objects to align that can be positioned in an m-by-n grid.

For example, the following figure contains six axes that have been placed approximately into two columns in the figure.



To align all axes in a grid, select each axes (**Shift**+click each one), then select **Smart Align and Distribute** from the **Tools** menu.

The resulting alignment and distribution of the axes are shown below.



Snap to Grid — Aligning Objects on a Grid

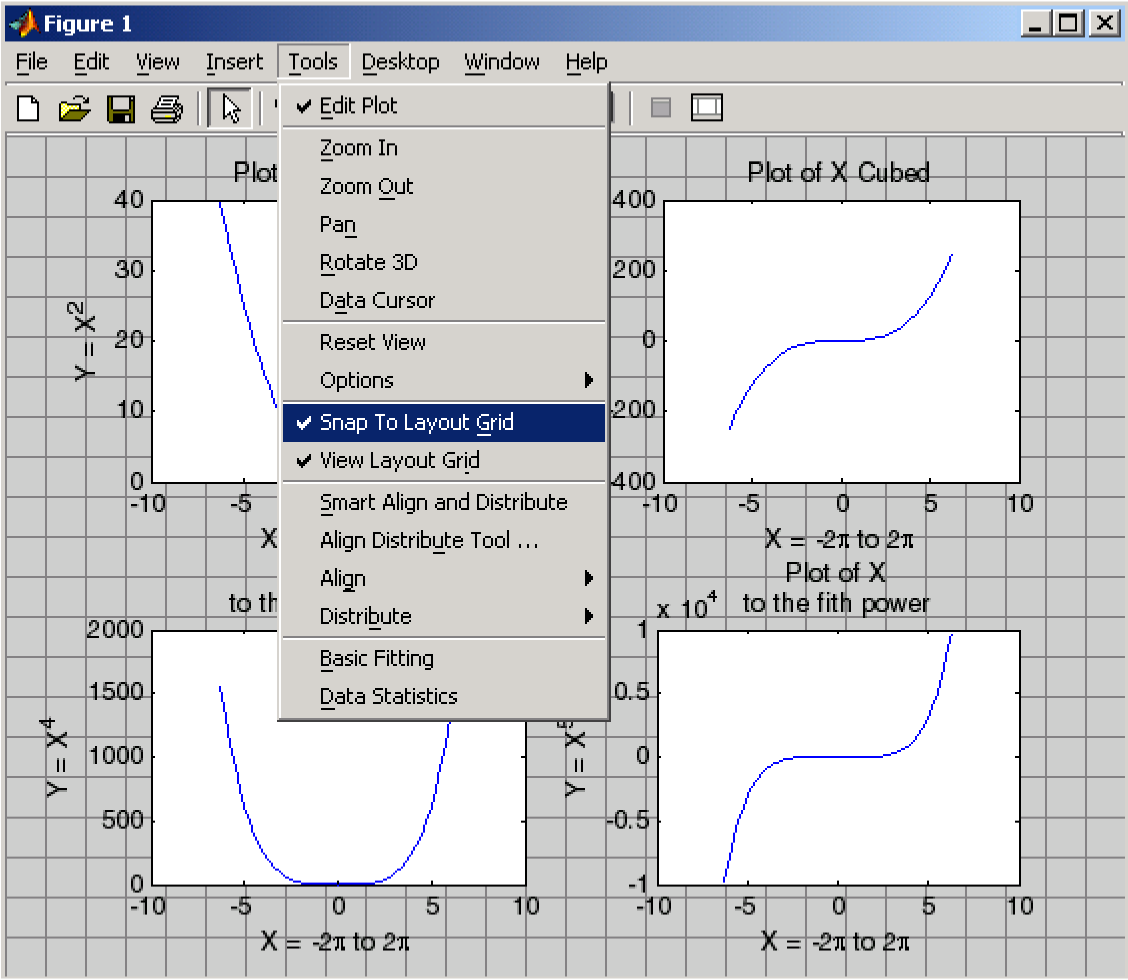
Figures have a layout grid that can aid the hand layout of objects displayed in the figure. You can also enable a snap-to-grid feature that forces objects to align with the grid increments when moved.

To display the grid on the figure background, select **View Layout Grid** from the **Tools** menu.

To force objects to align with the grid, select **Snap To Layout Grid** from the **Tools** menu.

To move objects in the figure, enable Plot Edit mode by selecting **Edit Plot** from the **Tools** menu. Click to select an object and then drag it to the desired location.

The following picture illustrates a figure with four subplots. You can select any of the four axes and move them. All axis labels and the title move with the axes. Annotation objects move independently of the plot axes.

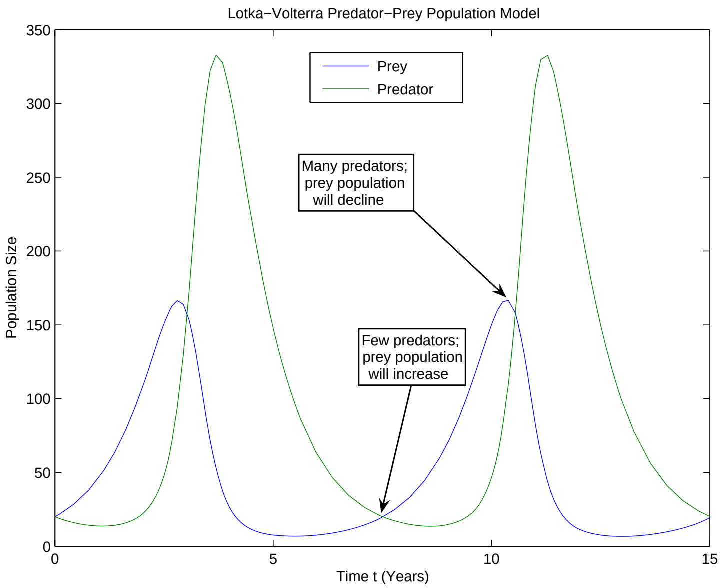

Figure 1
File Edit View Insert Tools Desktop Window Help
Edit Plot
Zoom In
Zoom Out
Pan
Rotate 3D
Data Cursor
Reset View
Options
Snap To Layout Grid
View Layout Grid
Smart Align and Distribute
Align Distribute Tool ...
Align
Distribute
Basic Fitting
Data Statistics
Plot of X Cubed
X = -2π to 2π
Plot of X
to the fith power
x 10^4
X = -2π to 2π
X = -2π to 2π

Adding Titles to Graphs

In this section...

What Is a Title?

In a MATLAB figure, a title is a text string at the top of an axes. It appears in the figure border, not within the axes it describes. Titles typically define the subject of the graph. The following figure shows a title, centered at its top.



Note While you can use text annotations to create a title for your graph, it is not recommended. Titles are anchored to the top of the axes they describe; text annotations are not. If you move or resize your axes, the title remains at the top. Additionally, if you cut a title and then paste it back into a figure, the title is no longer anchored to the axes.

You can add a title to a graph in several ways, described in the following sections.

Using the Title Option on the Insert Menu

To add a title to a graph using the **Insert** menu,

1 Click the **Insert** menu in the figure menu bar and choose **Title**. A text entry box opens at the top of the axes.

Note Selecting the **Title** option enables plot editing mode automatically.

---

2 Enter the text of the label.

3 When you are finished entering text, click anywhere in the figure background to close the text entry box around the title. If you click on another object in the figure, such as an axes or line, you close the title text entry box and also automatically select the object you clicked.

To change the font used in the title to bold, you must edit the title. You can edit the title as you would any other text object in a graph.

Using the Property Editor to Add a Title

To add a title to a graph using the Property Editor,

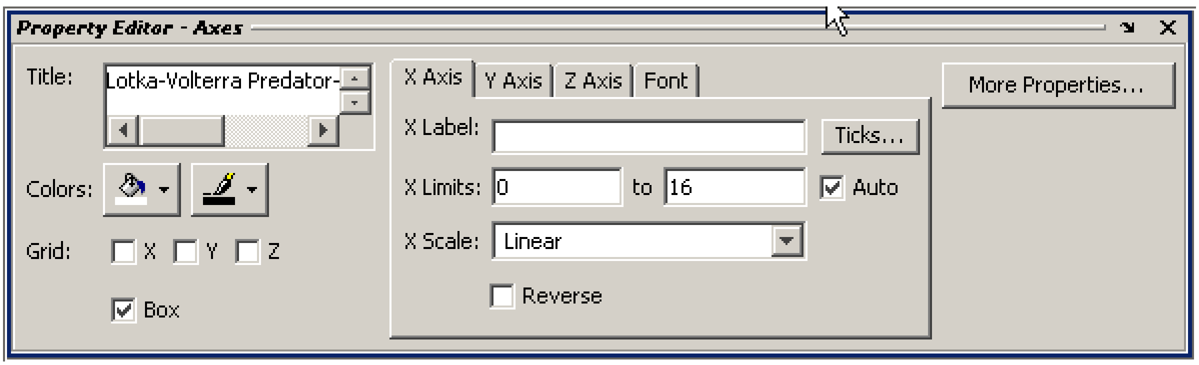
1 Start plot editing mode by selecting **Edit Plot** from the figure **Tools** menu.

2 Double-click an empty region of the axes in the graph. This starts the Property Editor. You can also start the Property Editor by right-clicking on

the axes and selecting **Show Property Editor** from the context menu or by selecting **Property Editor** from the **View** menu.

The Property Editor displays a property panel specific to axes objects. Titles are a property of axes objects.

3 Type the text of your title in the **Title** text entry box.



You can change the font, font style, position, and many other aspects of the title format.

- To move the title, select the text and drag it to the desired position.
- To edit the text, double-click the title and type new characters.
- To change the font and other text properties, select the title and right-click to display the context menu.

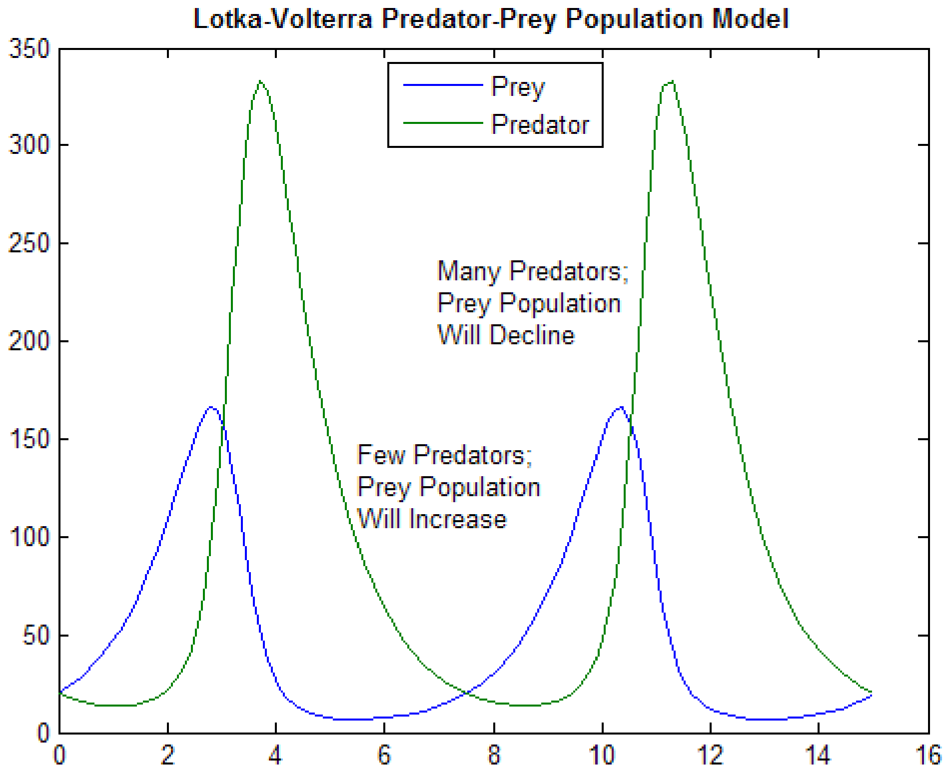
Using the title Function

To add a title to a graph at the MATLAB command prompt or from an M-file, use the `title` function.

For example, the following code adds a title to the current axes and sets the value of the `FontWeight` property to bold.

```
title('Lotka-Volterra Predator-Prey Population Model',...
  'FontWeight','bold')
```

The following figure shows a plot with this title. It also contains a legend and text objects.



Titles are associated with axes. This means that when you make subplots, each axes can have a distinct title. If you create a title in this way when no figures exist, a blank figure with an axes is generated displaying the title you specified.

To edit a title from the MATLAB command prompt or from an M-file, use the `set` function with the title's handle, as follows:

```
title_handle = title('This is the original title');
set(title_handle,'String','This is a revised title')
```

You can also change title strings in plot edit mode or with the Property Inspector. For more information, including code that generates the figure shown here, see the example in "Creating Text Annotations with the text or gtext Function" on page 3-46.

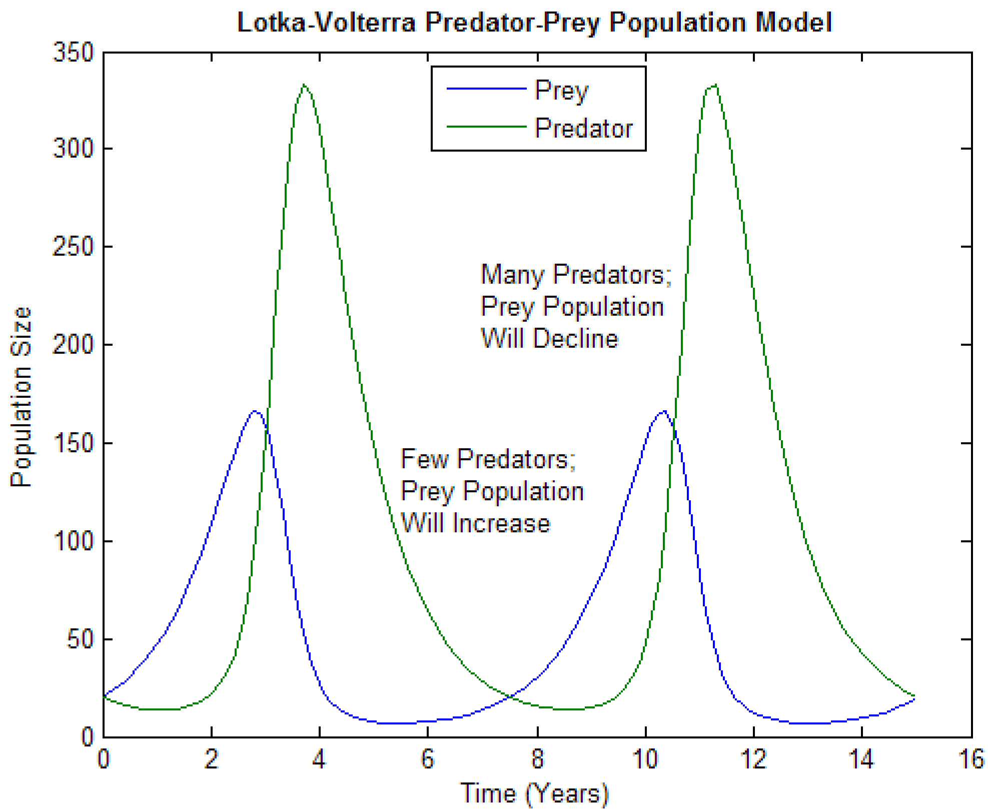
Adding Axis Labels to Graphs

In this section...

What Are Axis Labels?

In a MATLAB figure, an axis label is a text string aligned with the *x*-, y-, or *z*-axis in a graph. Axis labels can help explain the meaning of the units that each axis represents. The following figure shows axis labels for both axes, created at the Command Line using the `xlabel` and `ylabel` functions, as follows:

```
xlabel('Time (Years)')
ylabel('Population Size')
```



This is the simplest way to add axis labels using M-code, and is described in more detail in “Using Axis-Label Commands” on page 3-42 . You can add axis labels to a graph in other ways, as described in the following sections.

Note Although you can use free-form text annotations to create axes labels, it is not recommended. Axis labels are anchored to the axes they describe; text annotations are not. If you move or resize your axes, the labels automatically move with the axes. Additionally, if you cut a label and then paste it back into a figure, the label is no longer anchored to the axes.

Using the Label Options on the Insert Menu

1 Click the **Insert** menu and choose the label option that corresponds to the axis you want to label: **X Label**, **Y Label**, or **Z Label**. A text entry box opens along the axis or around an existing axis label.

Note Text editing boxes for the *y*- and *z*-axis labels are horizontal; the text you enter is automatically rotated to align the label with the axis when you finish entering text.

---

2 Enter the text of the label, or edit the text of an existing label.

3 Click anywhere else in the figure background to close the text entry box around the label. If you click on another object in the figure, such as an axes or line, you close the label text entry box but also automatically select the object you clicked.

Note After you use the **Insert** menu to add an axis label, plot edit mode is enabled in the figure, if it was not already enabled. You can modify axis labels in plot edit mode by double-clicking them and typing new text.

Using the Property Editor to Add Axis Labels

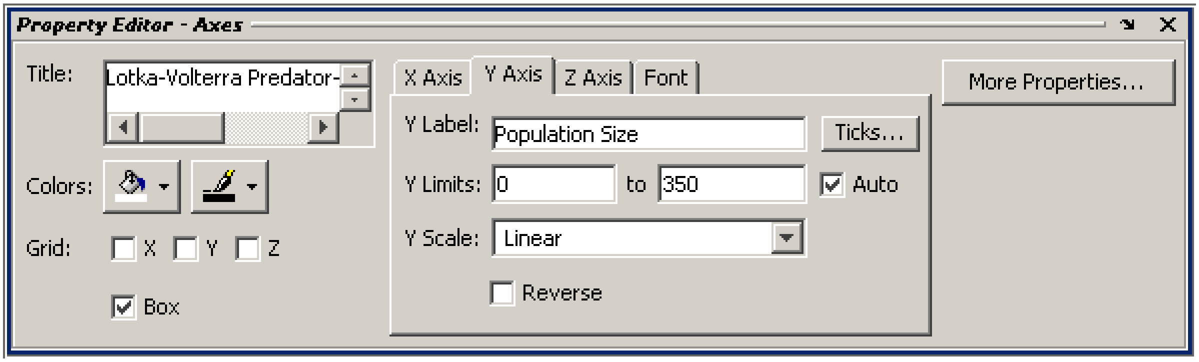
To add labels to a graph using the Property Editor,

1 Start plot editing mode by selecting **Edit Plot** from the figure **Tools** menu.

2 Start the Property Editor by double-clicking on the axes in the graph. You can also start the Property Editor by right-clicking on the axes and selecting **Properties** from the context menu or by selecting **Property Editor** from the **View** menu.

   The Property Editor displays the set of property panels specific to axes objects.

3 Select the **X Axis**, **Y Axis**, or **Z Axis** tab, depending on which axis label you want to add. Enter the label text in the text entry box.



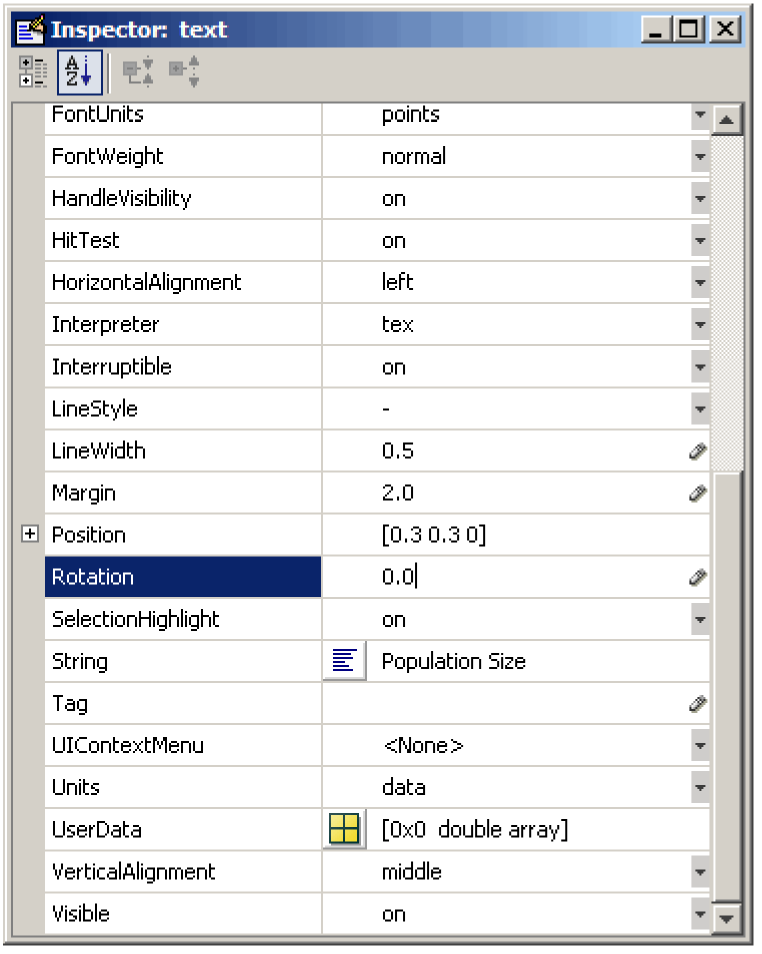
Rotating Axis Labels

You can rotate axis labels using the Property Editor:

1 Start plot editing mode by selecting **Edit Plot** from the figure **Tools** menu.

2 Display the Property Editor by selecting (left-clicking) the axis label you want to rotate. Right-click over the selected text, then choose **Properties** from the context menu.

3 Click the **More Properties** button to display the Property Inspector.



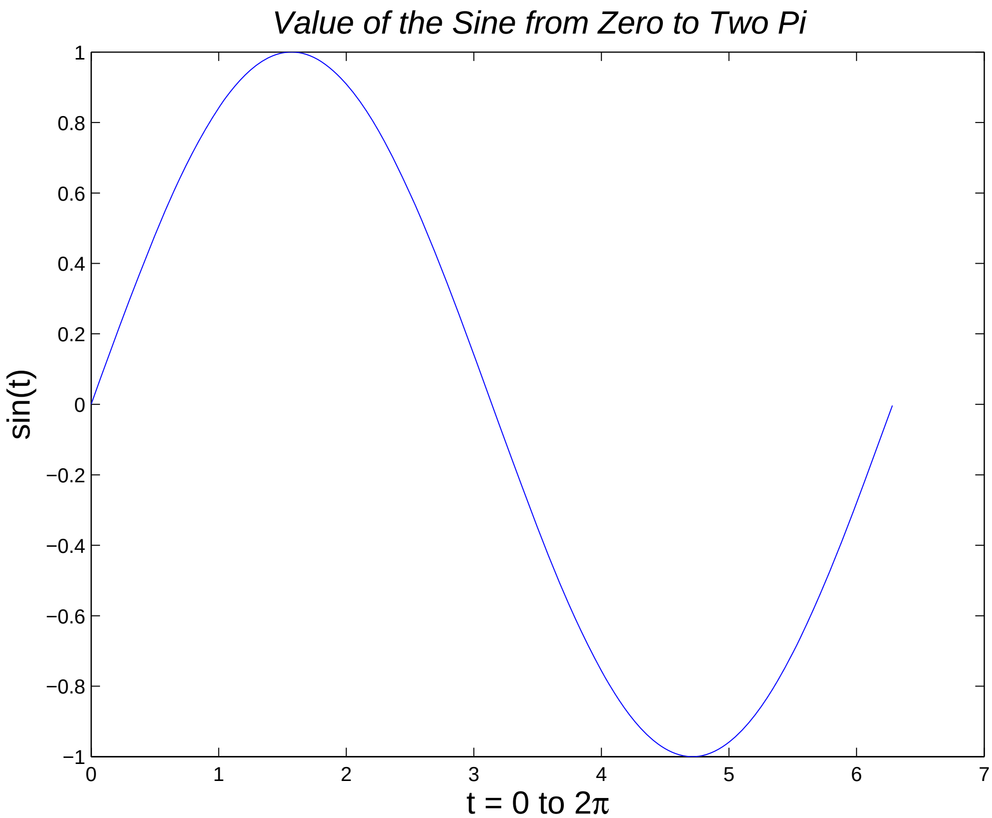
4 Select the **Rotation** property text field. A value of 0 degrees orients the label in the horizontal position.

5 With the left mouse button down on the selected label, drag the text to the desired location and release.

Using Axis-Label Commands

You can add *x*-, *y*-, and *z*-axis labels using the `xlabel`, `ylabel`, and `zlabel` functions. For example, these statements label the axes and add a title.

```
xlabel('t = 0 to 2\pi','FontSize',16)
ylabel('sin(t)','FontSize',16)
title('\it{Value of the Sine from Zero to Two Pi}','FontSize',16)
```



The labeling commands automatically position the text string appropriately. MATLAB interprets the characters immediately following the backslash (\) as TeX commands. These commands draw symbols such as Greek letters and arrows.

See the text `String` property for a list of TeX character sequences. See also the `texlabel` function for converting MATLAB expressions to TeX symbols.

Rotating Axis Labels Using Commands

Axis labels are text objects that you can rotate by specifying a value for the object's `Rotation` property. The handles of the *x*-, *y*-, and *z*-axis labels are stored in the axes `XLabel`, `YLabel`, and `ZLabel` properties respectively.

Therefore, to rotate the *y*-axis label so that the text is horizontal:

1 Get the handle of the text object using the axes `YLabel` property.

2 Set the `Rotation` property to 0.0 degrees.

For example, this statement rotates the text of the *y*-axis label on the current axes:

```
set(get(gca,'YLabel'),'Rotation',0.0)
```

Repositioning Axis Labels

You can reposition an axis label by dragging the text.

1 Start plot editing mode by selecting **Edit Plot** from the figure **Tools** menu.

2 Select the text of the label you want to reposition (handles appear around the text object).

3 With the left mouse button down on the selected label, drag the text to the desired location and release.

Adding Text Annotations to Graphs

In this section...

What Are Text Annotations?

Text annotations are boxes containing text strings that you compose. The box can have a border and a background, or be invisible. The text can be in any installed text font, and can include TeX or LaTeX markup. You can add free-form text annotations anywhere in a MATLAB figure to help explain your data or bring attention to specific points in your data sets.

As the following example shows, annotating a graph manually is easy in plot edit mode. When you enable plot editing, you can create text annotations by selecting the appropriate kind of annotation from the **Insert** menu, clicking in the graph or the figure background and then entering text. To insert **textarrow** annotations, you first drag out an arrow from tail to head, then type the text at the text cursor next to the tail,

You can also add text annotations from the command line, using the `text` or `gtext` function. The example illustrates how to use `text`.

Using plot editing mode or `gtext` makes it easy to place a text annotation where you want in a graph. Use the `text` function when you want to position

a text annotation at a specific point within an axes for which you know the coordinates.

Note Text annotations created using the `text` or `gtext` function are anchored to the axes. Text annotations created in plot edit mode are not. If you move or resize your axes, you will have to reposition your text annotations. For more information, see "Positioning Annotations in Data Space" on page 3-68.

Creating Text Annotations with the text or gtext Function

To create a text annotation using the `text` function, you must specify the text and its location within the axes, providing the *x*- and *y*-coordinates in the same `Units` that the graph uses (`pixels`, `normalized`, etc.).

Use the `gtext` function when you want to position a text annotation at a specific point in the data space with the mouse.

The following example adds text annotation, a title, and a legend to a graph of output from the Lotka-Volterra predator-prey population model. It also illustrates how to create multiline text annotations using cell arrays (also see the following section "Text in Cell Arrays" on page 3-57).

```
% Define initial conditions
t0 = 0;
tfinal = 15;
y0 = [20 20]';
% Simulate the differential equation
tfinal = tfinal*(1+eps);
[t,y] = ode23('lotka',[t0 tfinal],y0);
% Plot the two curves, storing handles to them
% so their DisplayNames can be set
hlines = plot(t,y);
% Compose and display two multiline text
% annotations as cell arrays
str1(1) = {'Many Predators;'};
str1(2) = {'Prey Population'};
str1(3) = {'Will Decline'};
```

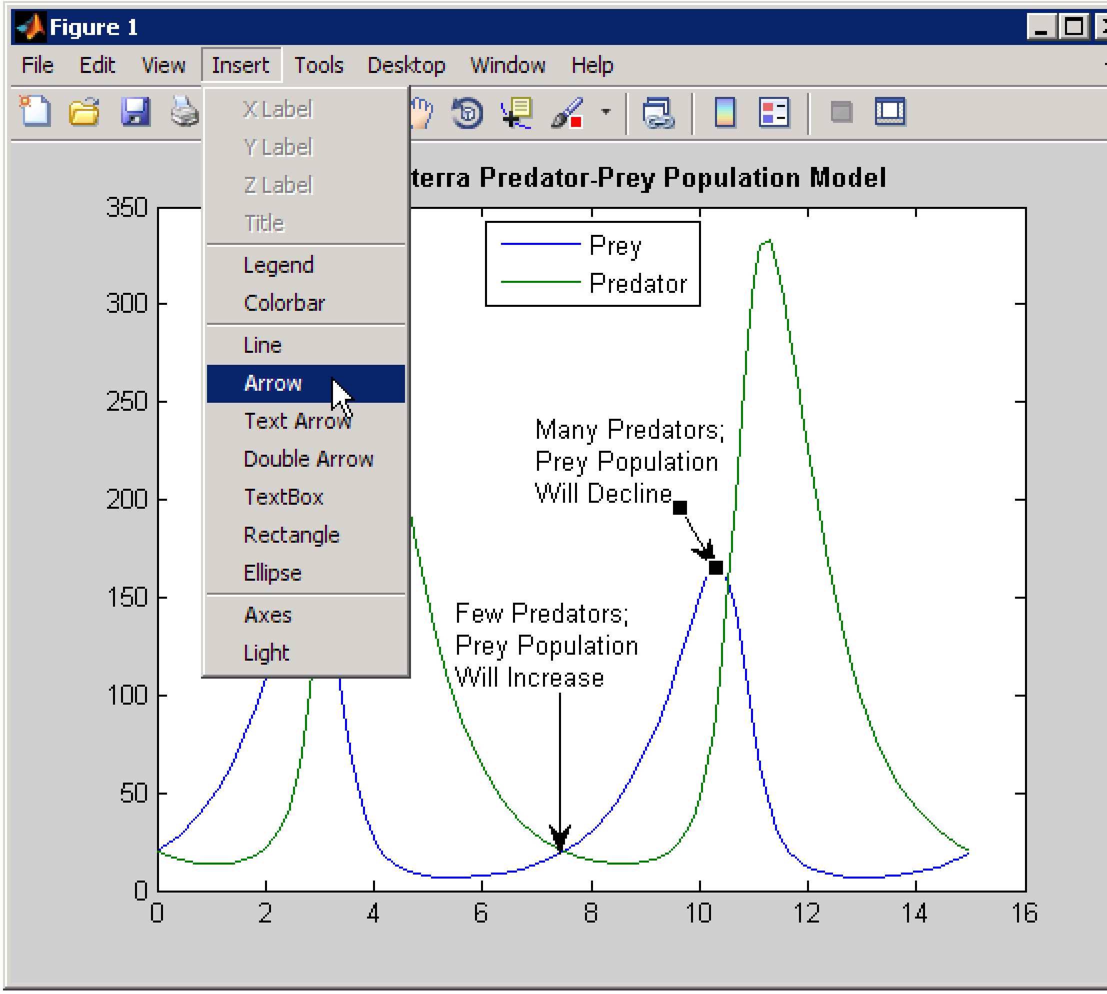
```
text(7,220,str1)
str2(1) = {'Few Predators;'};
str2(2) = {'Prey Population'};
str2(3) = {'Will Increase'};
text(5.5,125,str2)
% Set DisplayNames for the lines for use by the legend
set(hlines(1),'Displayname','Prey')
set(hlines(2),'Displayname','Predator')
% Center a legend at the top of the graph
legend('Location','north')
% Add a title with bold style
title('Lotka-Volterra Predator-Prey Population Model',...
  'FontWeight','bold')
```

To connect the text with the appropriate points on the plot, draw two annotation arrows by hand. First enter plot edit mode, either by typing

```
plotedit
```

in the Command Window or by clicking the **Edit Plot** icon  in the figure toolbar. (Type `plotedit` again or click the icon again when you want to exit plot edit mode.)

Select **Arrow** from the **Insert** menu. Draw an arrow from each block of text to point to the lines, as shown here.



Calculating the Positions of Text Annotations

You can also calculate the positions of text annotations in a graph. The following code adds annotations at three data points on a graph.

```
t=0:pi/64:2*pi;
plot(t,sin(t));
title('The Sine of 0 to 2\pi')
```

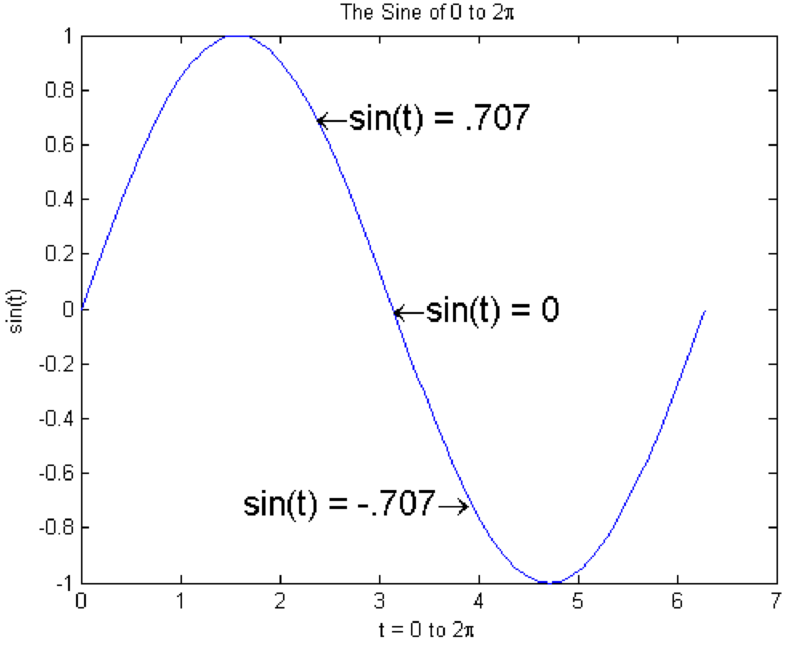
```
xlabel('t = 0 to 2\pi')
ylabel('sin(t)')

text(3*pi/4,sin(3*pi/4),...
     '\leftarrowsin(t) = .707',...
     'FontSize',16)

text(pi,sin(pi),'\leftarrowsin(t) = 0',...
     'FontSize',16)

text(5*pi/4,sin(5*pi/4),'sin(t) = -.707\rightarrow',...
     'HorizontalAlignment','right',...
     'FontSize',16)
```

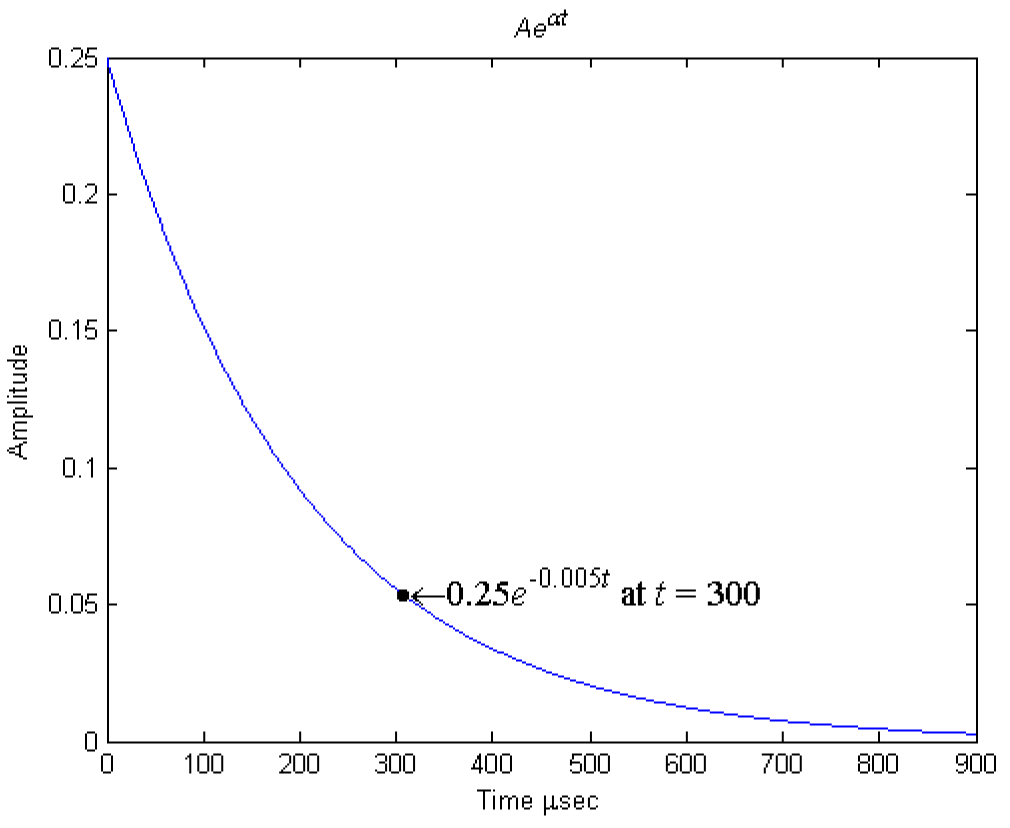
The `HorizontalAlignment` of the text string `'sin(t) = -.707 \rightarrow'`- is set to `right` to place it on the left side of the point `[5*pi/4,sin(5*pi/4)]` on the graph. For more information about aligning text annotations, see "Text Alignment" on page 3-51.



Defining Symbols. For information on using symbols in text strings, see "Mathematical Symbols, Greek Letters, and TEX Characters" on page 3-55.

You can use text objects to annotate axes at arbitrary locations. Text is positioned using the data units of the axes. For example, suppose you plot the function $y = Ae^{-\alpha t}$ with $A = 0.25$, $\alpha = 0.005$, and $t = 0$ to 900.

```
t = 0:900;
plot(t,0.25*exp(-0.005*t))
xlabel('Time \musec')
ylabel('Amplitude')
title('\itAe^\alpha^t')
```



To annotate the point where the value of $t = 300$, calculate the text coordinates using the function you are plotting.

```
text(300,.25*exp(-0.005*300),...
title('\itAe^\alpha^t')['\bullet\leftarrow\...
fontname{times}0.25{\ite}^{-0.005{\itt}}' ...
' at {\itt} = 300'],'FontSize',14)
```

This statement defines the text `Position` property as

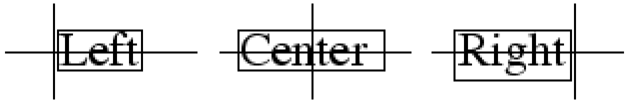
$x = 300,\ y = 0.25e^{-0.005 \times 300}$

The default text alignment places this point to the left of the string and centered vertically with the rectangle defined by the text `Extent` property. The following section provides more information about changing the default text alignment.

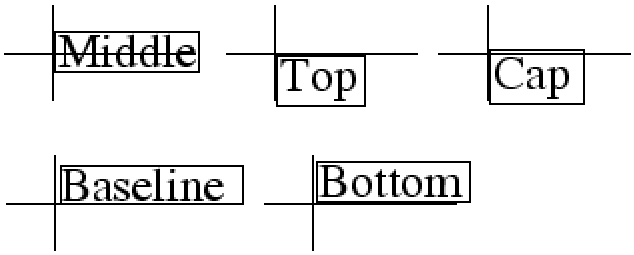
Text Alignment

The `HorizontalAlignment` and the `VerticalAlignment` properties control the placement of the text characters with respect to the specified *x*-, *y*-, and z-coordinates. The following diagram illustrates the options for each property and the corresponding placement of the text.

Text HorizontalAlignment property viewed with the VerticalAlignment property set to middle (the default).



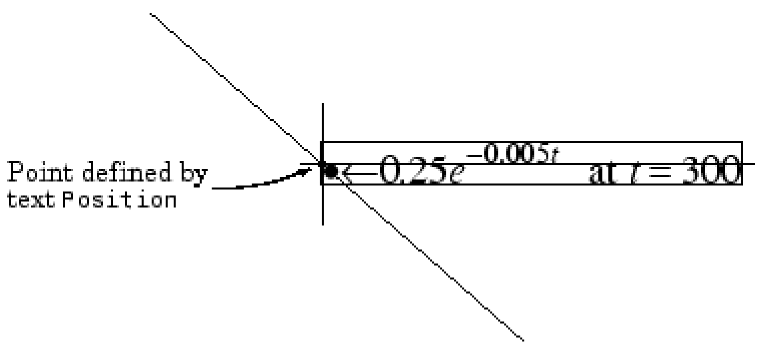
Text VerticalAlignment property viewed with the HorizontalAlignment property set to left (the default).



The default alignment is

- `HorizontalAlignment = 'left'`
- `VerticalAlignment = 'middle'`

The text `String` is not placed exactly on the specified `Position`. For example, the previous section showed a plot with a point annotated with text. Zooming in on the plot enables you to see the actual positioning of the text.



The small dot is the point specified by the text `Position` property. The larger dot is the bullet defined as the first character in the text `String` property.

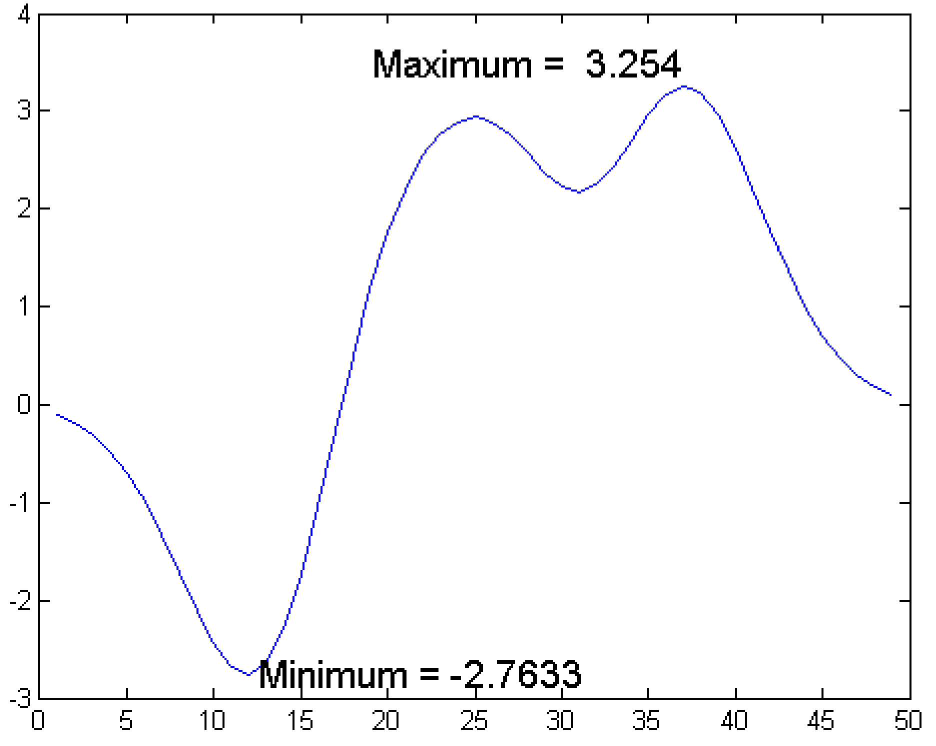
Example — Aligning Text

Suppose you want to label the minimum and maximum values in a plot with text that is anchored to these points and that displays the actual values. This example uses the plotted data to determine the location of the text and the values to display on the graph. One column from the `peaks` matrix generates the data to plot.

```
Z = peaks;
h = plot(Z(:,33));
```

The first step is to find the indices of the minimum and maximum values to determine the coordinates needed to position the text at these points (`get`, `find`). Then create the string by concatenating the values with a description of what the values are.

```
x = get(h,'XData');   % Get the plotted data
y = get(h,'YData');
imin = find(min(y) == y);  % Find the index of the min and max
imax = find(max(y) == y);
text(x(imin),y(imin),[' Minimum = ',num2str(y(imin))],...
 'VerticalAlignment','middle',...
 'HorizontalAlignment','left',...
 'FontSize',14)
text(x(imax),y(imax),['Maximum =  ',num2str(y(imax))],...
 'VerticalAlignment','bottom',...
 'HorizontalAlignment','right',...
 'FontSize',14)
```



The `text` function positions the string relative to the point specified by the coordinates, in accordance with the settings of the alignment properties. For the minimum value, the string appears to the right of the text position point; for the maximum value the string appears above and to the left of the text position point. The text always remains in the plane of the computer screen, regardless of the view.

Editing Text Objects

You can edit any of the text labels or annotations in a graph:

1 Start plot edit mode.

2 Double-click the string, or right-click the string and select **Edit** from the context menu.

An editing bar (|) appears next to the text.

3 Make any changes to the text.

4 Click anywhere outside the text edit box to end text editing.

Note To create special characters in text, such as Greek letters or mathematical symbols, use TEX sequences. See the text `string` property for a table of characters you can use. If you create special characters by using the **Font** dialog box (available via text objects' context menus, and also found in the Property Editor) and selecting the Symbol font family, you cannot edit that text object using MATLAB commands.

Mathematical Symbols, Greek Letters, and TEX Characters

You can include mathematical symbols and Greek letters in text using TEX-style character sequences. This section describes how to construct a TEX character sequence.

Two Levels of MATLAB TEX Support

There are two levels of TEX support, controlled by the text `Interpreter` property:

- `'tex'` — Support for a subset of TEX markup
- `'latex'` — Support for TEX and LATEX markup

If you do not want the characters interpreted as TEX markup, then set the `interpreter` property to `'none'`.

Available Symbols and Greek Letters

For a list of symbols and the character sequences used to define them, see the table of available TEX characters in the Text Properties reference page.

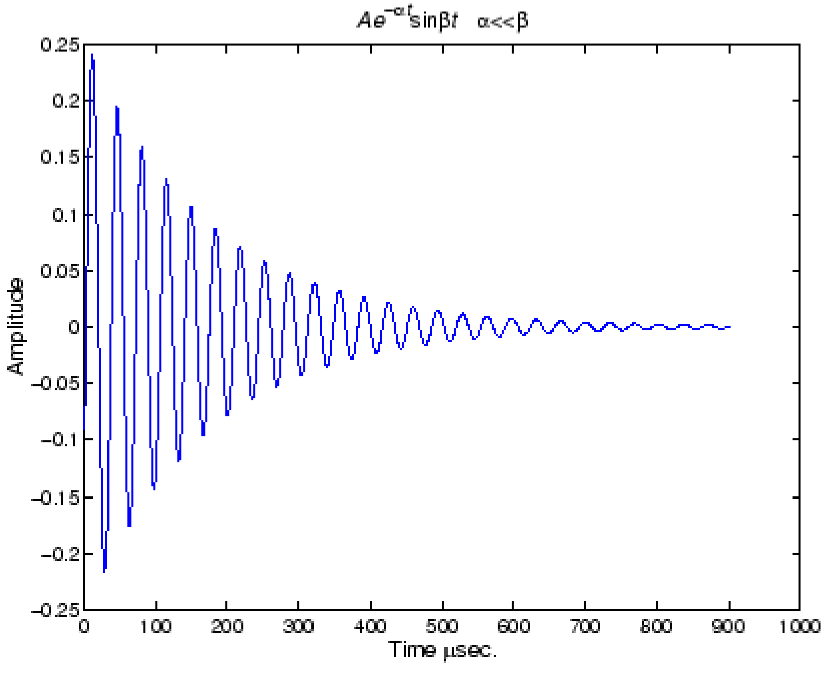
In general, you can define text that includes symbols and Greek letters using the `text` function, assigning the character sequence to the `String` property

of text objects. You can also include these character sequences in the string arguments of the `title`, `xlabel`, `ylabel`, and `zlabel` functions.

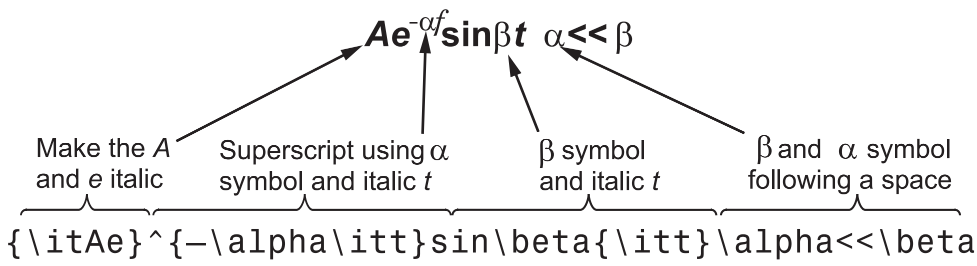
Example — Using a Mathematical Expression to Title a Graph

This example uses TEX character sequences to create graph labels. The following statements add a title and *x*- and *y*-axis labels to an existing graph.

```
title('{\itAe}^{-\alpha\itt}sin\beta{\itt} \alpha<<\beta')
xlabel('Time \musec.')
ylabel('Amplitude')
```



The backslash character (\) precedes all TEX character sequences. Looking at the string defining the title illustrates how to use these characters.



Controlling the Interpretation of TEX Characters

The text `Interpreter` property controls the interpretation of TEX characters. If you set this property to `none`, MATLAB interprets the special characters literally.

Using Character and Numeric Variables in Text

Any string variable is a valid specification for the text `String` property. This section illustrates how to use matrix, cell array, and numeric variables as arguments to the `text` function.

Text in Character Arrays

For example, each row of the matrix `PersonalData` contains specific information about a person, padding all but the longest row with a space so that each has the same number of columns).

```
PersonalData = ['Jack Straw ';'489 Main St';'Wichita KS '];
```

To display the data, index into the desired row.

```
text(.3,.5,['Name: ',PersonalData(1,:)])
text(.3,.45,['Address: ',PersonalData(2,:)])
text(.3,.4,['City and State: ',PersonalData(3,:)])
```

Text in Cell Arrays

Using a cell array enables you to create multiline text with a single text object. Each cell does not need to be the same number of characters. For example, the following statements,

```
key(1)={'{\itAe}^{-\alpha\itt}sin\beta{\itt}'};
key(2)={'Time in \musec'};
key(3)={'Amplitude in volts'};
text(.1,.8,key)
```

produce this output.

$Ae^{-\alpha t}\sin\beta t$
Time in μsec
Amplitude in volts

Numeric Variables

You can specify numeric variables in text strings using the `num2str` (number to string) function. For example, if you type on the command line

```
x = 21;
['Today is the ',num2str(x),'st day.']
```

The three separate strings concatenate into one.

```
Today is the 21st day.
```

Since the result is a valid string, you can specify it as a value for the text `String` property.

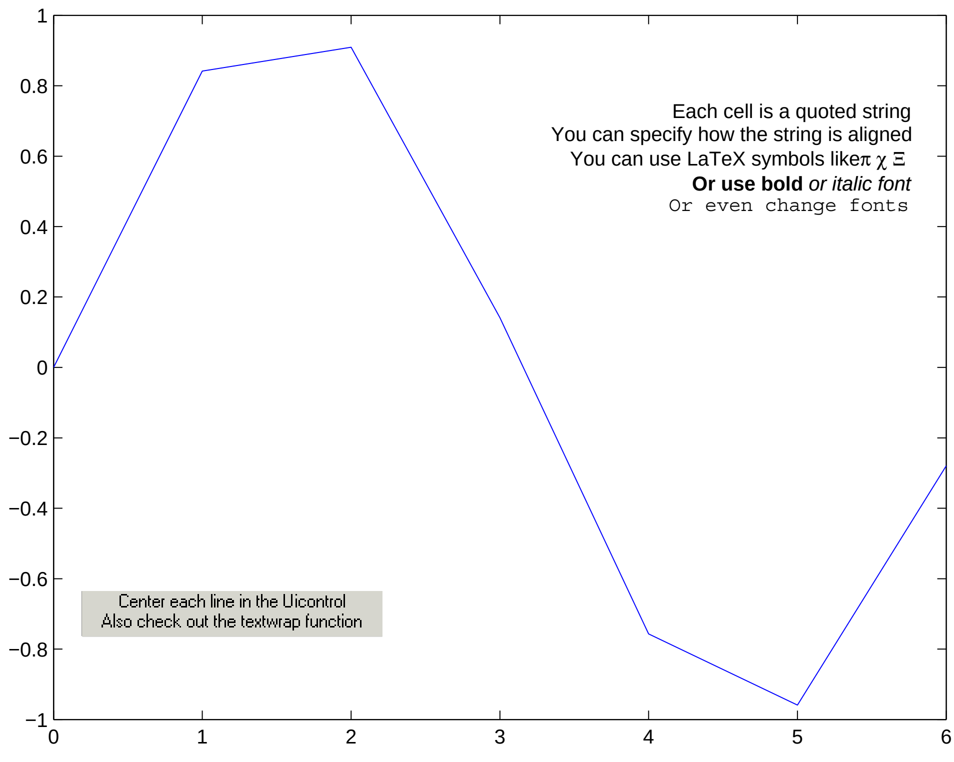
```
text(xcoord,ycoord,['Today is the ',num2str(x),'st day.'])
```

Example — Multiline Text

You can input multiline text strings using cell arrays. Simply define a string variable as a cell array with one line per cell. This example defines two cell arrays, one used for a `uicontrol` and the other as `text`.

```
str1(1) = {'Center each line in the Uicontrol'};
str1(2) = {'Also check out the textwrap function'};
str2(1) = {'Each cell is a quoted string'};
str2(2) = {'You can specify how the string is aligned'};
str2(3) = {'You can use LaTeX symbols like \pi \chi \Xi'};
str2(4) = {'\bfOr use bold \rm\itor italic font\rm'};
str2(5) = {'\fontname{courier}Or even change fonts'};
```

```
plot(0:6,sin(0:6))
uicontrol('Style','text','Position',[80 80 200 30],...
          'String',str1);
text(5.75,sin(2.5),str2,'HorizontalAlignment','right')
```



Example — Using LaTeX to Format Math Equations

The LaTeX markup language evolved from TEX, and has a superset of its capabilities. LaTeX gives you more elaborate control over specifying and styling mathematical symbols.

The following example illustrates some LaTeX typesetting capabilities when used with the `text` function. Because the default interpreter is for TeX, you need to specify the parameter-value pair `'interpreter','latex'` when typesetting equations such as are contained in the following script:

```
%% LaTeX Examples--Some well known equations rendered in LaTeX
%
figure('color','white','units','inches','position',[2 2 4 6.5]);
axis off

%% A matrix; LaTeX code is
% \hbox {magic(3) is } \left( {\matrix{ 8 & 1 & 6 \cr
% 3 & 5 & 7 \cr 4 & 9 & 2 } } \right)
h(1) = text('units','inch', 'position',[.2 5], ...
    'fontsize',14, 'interpreter','latex', 'string',...
    ['$$\hbox {magic(3) is } \left( {\matrix{ 8 & 1 & 6 \cr'...
    '3 & 5 & 7 \cr 4 & 9 & 2 } } \right)$$']);

%% A 2-D rotation transform; LaTeX code is
%  \left[ {\matrix{\cos(\phi) & -\sin(\phi) \cr
%  \sin(\phi) & \cos(\phi) \cr}}
%  \right] \left[ \matrix{x \cr y} \right]
%
%  $$ \left[ {\matrix{\cos(\phi)
%  & -\sin(\phi) \cr \sin(\phi) & \cos(\phi)  % \cr}}
%  \right] \left[ \matrix{x \cr y} \right] $$
%
h(2) = text('units','inch', 'position',[.2 4], ...
    'fontsize',14, 'interpreter','latex', 'string',...
    ['$$\left[ {\matrix{\cos(\phi) & -\sin(\phi) \cr'...
    '\sin(\phi) & \cos(\phi) \cr}} \right]'...
    '\left[ \matrix{x \cr y} \right]$$']);

%% The Laplace transform; LaTeX code is
%  L\{f(t)\}  \equiv  F(s) = \int_0^\infty\!\!{e^{-st}f(t)dt}
%  $$ L\{f(t)\} \equiv  F(s) = \int_0^\infty\!\!{e^{-st}f(t)dt} $$
%  The Initial Value Theorem for the Laplace transform:
%  \lim_{s \rightarrow \infty} sF(s) = \lim_{t \rightarrow 0} f(t)
%  $$ \lim_{s \rightarrow \infty} sF(s) = \lim_{t \rightarrow 0}
%  f(t) $$
```

```
%
h(3) = text('units','inch', 'position',[.2 3], ...
    'fontsize',14, 'interpreter','latex', 'string',...
    ['$$L\{f(t)\}  \equiv  F(s) = \int_0^\infty\!\!{e^{-st}'...
    'f(t)dt}$$']);

%% The definition of e; LaTeX code is
%  e = \sum_{k=0}^\infty {1 \over {k!} }
%  $$ e = \sum_{k=0}^\infty {1 \over {k!} } $$
%
h(4) = text('units','inch', 'position',[.2 2], ...
    'fontsize',14, 'interpreter','latex', 'string',...
    '$$e = \sum_{k=0}^\infty {1 \over {k!} } $$');

%% Differential equation
% The equation for motion of a falling body with air resistance
% LaTeX code is
%  m \ddot y = -m g + C_D \cdot {1 \over 2} \rho {\dot y}^2 \cdot A
%  $$ m \ddot y = -m g + C_D \cdot {1 \over 2} \rho {\dot y}^2
%  \cdot A  $$
%
h(5) = text('units','inch', 'position',[.2 1], ...
    'fontsize',14, 'interpreter','latex', 'string',...
    ['$$m \ddot y = -m g + C_D \cdot {1 \over 2}'...
    '\rho {\dot y}^2 \cdot A$$']);

%% Integral Equation; LaTeX code is
%  \int_{0}^{\infty} x^2 e^{-x^2} dx = \frac{\sqrt{\pi}}{4}
%  $$ \int_{0}^{\infty} x^2 e^{-x^2} dx = \frac{\sqrt{\pi}}{4} $$
%
h(6) = text('units','inch', 'position',[.2 0], ...
    'fontsize',14, 'interpreter','latex', 'string',...
    '$$\int_{0}^{\infty} x^2 e^{-x^2} dx = \frac{\sqrt{\pi}}{4}$$');
```

$$\text{magic(3) is } \begin{pmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{pmatrix}$$

$$\begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$L\{f(t)\} \equiv F(s) = \int_0^\infty e^{-st} f(t) dt$$

$$e = \sum_{k=0}^\infty \frac{1}{k!}$$

$$m\ddot{y} = -mg + C_D \cdot \frac{1}{2}\rho\dot{y}^2 \cdot A$$

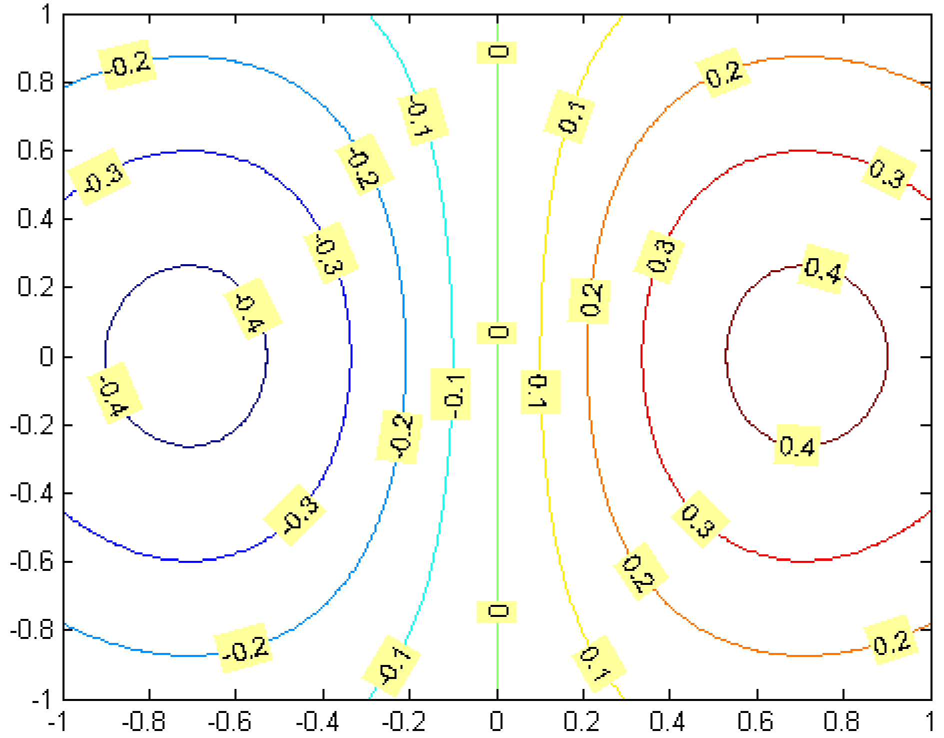
$$\int_0^\infty x^2 e^{-x^2} dx = \frac{\sqrt{\pi}}{4}$$

You can find out more about the LaT_eX system at The LaTeX Project Web site, http://www.latex-project.org/.

Drawing Text in a Box

When you use the `text` function to display a character string, the string's position is defined by a rectangle called the `Extent` of the text. You can display this rectangle either as a box or a filled area. For example, you can highlight contour labels to make the text easier to read.

```
[x,y] = meshgrid(-1:.01:1);
z = x.*exp(-x.^2-y.^2);
[c,h]=contour(x,y,z);
h = clabel(c,h);
set(h,'BackgroundColor',[1 1 .6])
```



For additional features, see the following text properties:

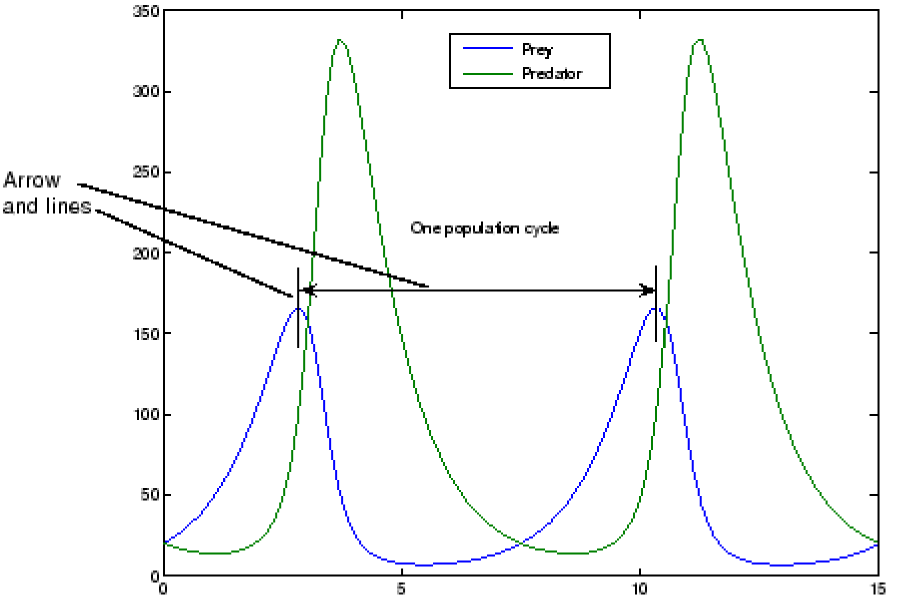
- `BackgroundColor` — Color of the rectangle's interior (`'none'` by default)
- `EdgeColor` — Color of the rectangle's edge (`'none'` by default)
- `LineStyle` — Style of the rectangle's edge line (first set `EdgeColor`)
- `LineWidth` — Width of the rectangle's edge line (first set `EdgeColor`)
- `Margin` — Increase the size of the rectangle by adding a margin to the text extent.

Adding Arrows and Lines to Graphs

In this section...

Creating Arrows and Lines in Plot Editing Mode

With plot editing mode enabled, you can add arrows and lines anywhere in a figure window.



You can also use arrow characters (TeX characters) to create arrows using the `text` function. However, arrows created this way can only point to the left or right, horizontally. See "Calculating the Positions of Text Annotations" on page 3-48 for an example.

To add an arrow or line annotation to a graph,

1 Click the **Insert** menu and choose the **Arrow** or **Line** option, or click the **Arrow** or **Line** button in the Plot Edit toolbar.

   The cursor changes to a cross-hair.

2 Position the cursor in the figure where you want to start the line or arrow and press either mouse button. Hold the button down and move the mouse to define the length and direction of the line or arrow.

3 Release the mouse button.

Note After you add an arrow or line, plot edit mode is enabled in the figure, if it was not already enabled.

Editing Arrows and Line Annotations

You can edit the appearance of arrow and line annotations using the context menu.

With plot editing mode enabled, right-click the arrow or line annotation to display its context menu.



You can select an annotation and then choose **Show M-code** to obtain a code snippet that you can insert in a function or script to reproduce the annotation.

For more options, select **Properties** to display the Property Editor.

Positioning Annotations in Data Space

Example — Pinning Text Arrows and Ellipses

Annotation objects (`arrow`, `doublearrow`, `textarrow`, `ellipse`, `line`, `rectangle`, and `textbox`) are attached to figures rather than to axes. By default, they have normalized figure coordinates when first created. For information about figure coordinates, see "Positioning Figures" on page 9-6. Although this enables you to place annotation objects anywhere within a figure, it also makes it difficult to precisely locate them on graphs so that they relate to data space coordinates (the *x*,*y* units of plotted data). You can position annotations in data space by transforming the data space coordinates to normalized figure coordinates. This example shows how to do this using a function called `dsxy2figxy`, which is an M-file that accompanies this documentation. (It is *not* a MATLAB function.) You specify data space coordinates that an annotation object should occupy as arguments to the function. The function returns the figure coordinates where the annotation should be placed.

1 Make the function `dsxy2figxy` available to you in one of the following ways:

- Click here to open function `dsxy2figxy` in the MATLAB editor, and then save it to your current directory or elsewhere on the MATLAB path.
- Execute the following command to add the directory where function `dsxy2figxy` is stored to the MATLAB path.

```
addpath([docroot '/techdoc/creating_plots/examples'])
```

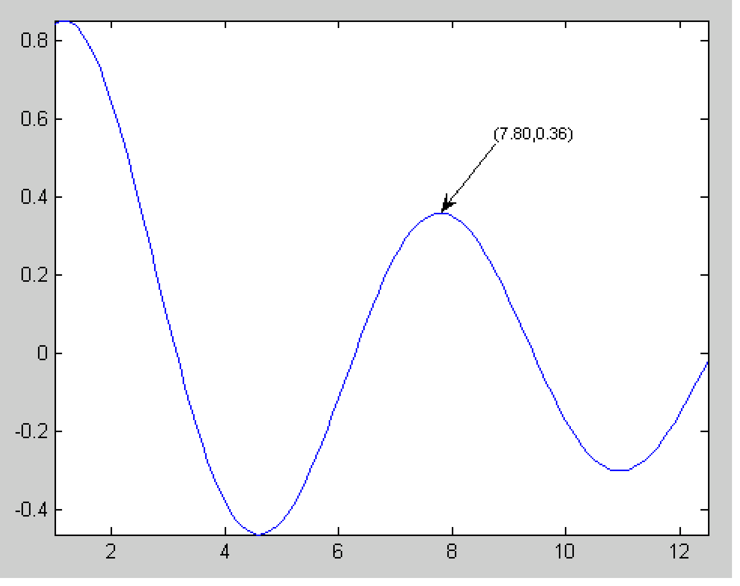
1 Create sine function data and make a line plot of it:

```
x1 = 1:.1:4*pi;
y1 = sin(x1)./sqrt(x1);
figure
plot(x1,y1)
axis tight
```

2 Interactively place a text arrow on the graph with `ginput`, which places a cross-hair cursor on the axes for users to select point locations. This function returns *x*,*y* coordinate pairs in data space.

You can use `ginput` to interactively locate a text arrow annotation. When called as follows, it accepts two clicks before exiting:

```
disp('Click graph to place arrow; first tail, then head:')
[axx axy] = ginput(2);   % Returns list of x, list of y in data space
% Transform from data space to figure space
[arrowx,arrowy] = dsxy2figxy(gca, axx, axy);
har = annotation('textarrow',arrowx,arrowy);
content = sprintf('(%4.2f,%4.2f)',axx(2), axy(2));
% Plot anno text centered at the tail of the arrow
set(har,'String',content,'Fontsize',8)
```

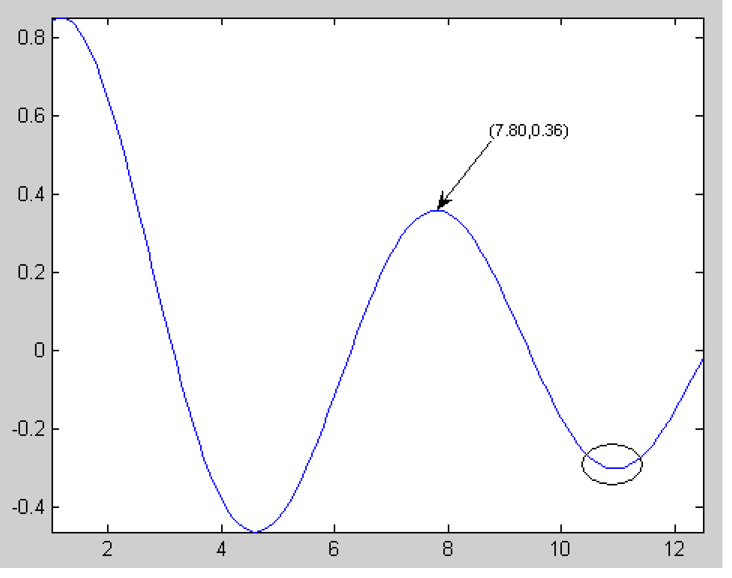


3 Now place an ellipse on the axes

To place ellipses, you need a coordinate box (position rectangle) instead of two *x*,*y* tuples. The function `dsxy2figxy` computes and returns a position rectangle if it is called with one:

```
disp('Click in the axes to define the bounding box of an ellipse:')
[axx axy] = ginput(2);      % Returns list of x, list of y in data space
abox(1) = min(axx); abox(2) = min(axy);     % Get minimum x and y coords
```

```
abox(3) = abs(axx(1)-axx(2));               % Get box width
abox(4) = abs(axy(1)-axy(2));               % Get box height
% Transform from data space to figure space
[bbox] = dsxy2figxy(gca, abox);
% Plot the ellipse where you clicked
annotation('ellipse',bbox);
```



Here is the M-help for `dsxy2figxy`.

```
dsxy2figxy -- Transform point or position from data space
coordinates into normalized figure coordinates
Transforms [x y] or [x y width height] vectors from data space
coordinates to normalized figure coordinates in order to locate
annotation objects within a figure. These objects are: arrow,
doublearrow, textarrow, ellipse, line, rectangle, textbox

Syntax:
   [figx figy] = dsxy2figxy([x1 y1],[x2 y2])  % GCA is used
   figpos      = dsxy2figxy([x1 y1 width height])
   [figx figy] = dsxy2figxy(axes_handle, [x1 y1],[x2 y2])
   figpos      = dsxy2figxy(axes_handle, [x1 y1 width height])
```

```
Usage: Obtain a position on a plot in data space and
       apply this function to locate an annotation there, e.g.,
  [axx axy] = ginput(2); (input is in data space)
  [figx figy] = dsxy2figxy(gca, axx, axy);  (now in figure space)
  har = annotation('textarrow',figx,figy);
  set(har,'String',['(' num2str(axx(2)) ',' num2str(axy(2)) ')'])

  Copyright 2006-2009 The MathWorks, Inc.
```

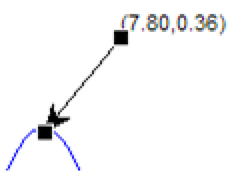
If you resize the figure, the annotations can change shape but continue to point to the same locations on the graph. This is because they and the figure use normalized coordinates. However, if you shift the axes up, down, left, or right within the figure—as you can in plot edit mode—the annotations remain fixed in figure space and do not move with the axes. The following section explains how to ensure that annotations stay connected to the data with which you have associated them.

Anchoring Annotations to Data Points

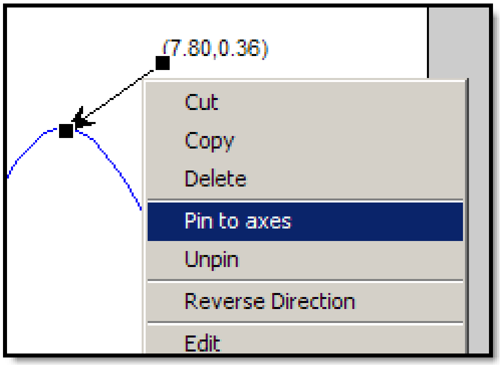
To enable annotations to remain anchored when you reposition axes (for example, when panning across the axes), you can manually pin them to locations on data graphs:

1 Enter plot edit mode by pushing the arrow button  on the figure toolbar.

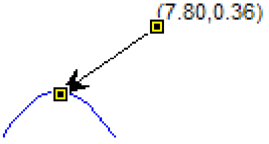
2 Click the arrow you placed on the graph to select it.



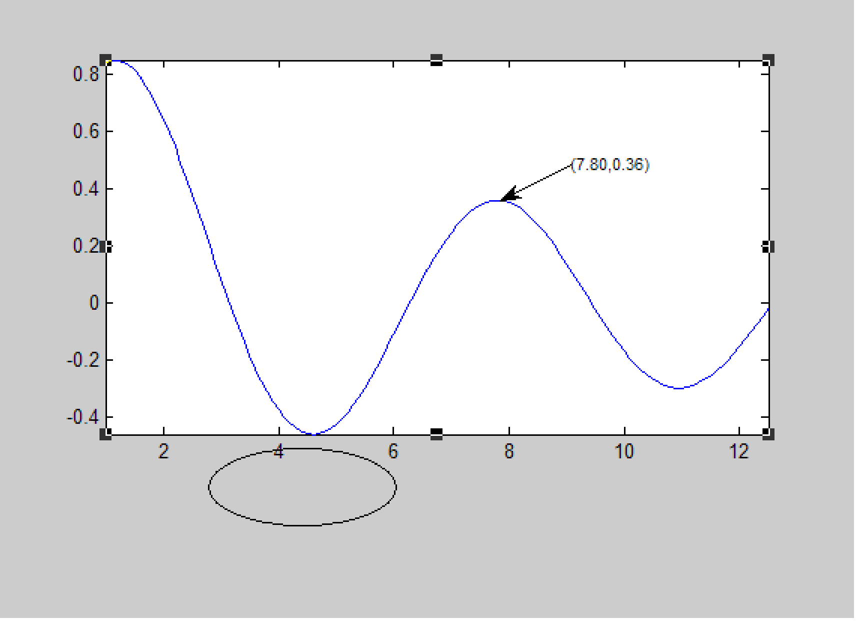
3 Right-click the arrow and choose **Pin to axes** from the context menu.



The black handles of the arrow become hollow to indicate that the object has been pinned.



4 Now when you change the position or shape of the axes, the arrow remains attached to the graph, but the oval does not.



The result of pinning annotations manually is functionally the same as computing locations for them in data space, except it requires user interaction and registers the annotation to the axes instead of to the figure. For more information, see "Pinning the Arrowhead End" on page 3-12, "Pinning Rectangles and Ellipses" on page 3-6, and "Pinning the Textbox" on page 3-9.

Basic Plotting Commands

Setting Up Figures

In this section...

Creating Figure Windows

MATLAB graphics are directed to a window that is separate from the Command Window. This window is referred to as a *figure*. The characteristics of this window are controlled by your computer's windowing system and MATLAB figure properties (see a description of each property). See Chapter 9, "Figure Properties" for some examples illustrating how to use figure properties.

Graphics functions automatically create new MATLAB figure windows if none currently exist. If a figure already exists, that window is used. If multiple figures exist, one is designated as the *current figure* and is used (this is generally the last figure used or the last figure you clicked the mouse in).

The `figure` function creates figure windows. For example,

```
figure
```

creates a new window and makes it the current figure. You can make an existing figure current by clicking it with the mouse or by passing its handle (the number indicated in the window title bar), as an argument to `figure`.

```
figure(h)
```

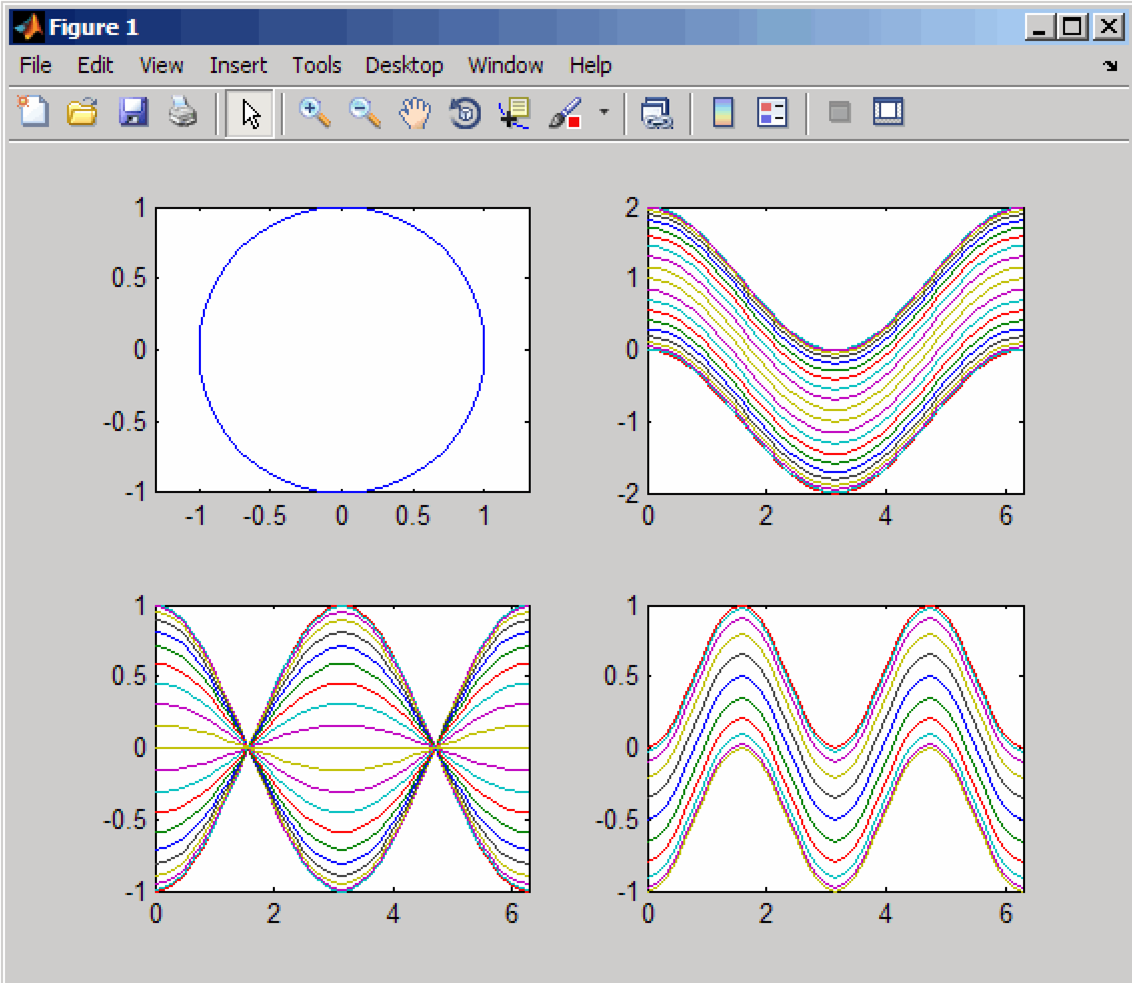
Displaying Multiple Plots per Figure

You can display multiple plots in the same figure window and print them on the same piece of paper with the `subplot` function.

`subplot(m,n,i)` breaks the figure window into an *m*-by-*n* matrix of small subplots and selects the `i`the subplot for the current plot. The plots are numbered along the top row of the figure window, then the second row, and so forth.

For example, the following statements plot data in four different subregions of the figure window.

```
t = 0:pi/20:2*pi;
[x,y] = meshgrid(t);
subplot(2,2,1)
plot(sin(t),cos(t))
axis equal
subplot(2,2,2)
z = sin(x)+cos(y);
plot(t,z)
axis([0 2*pi -2 2])
subplot(2,2,3)
z = sin(x).*cos(y);
plot(t,z)
axis([0 2*pi -1 1])
subplot(2,2,4)
z = (sin(x).^2)-(cos(y).^2);
plot(t,z)
axis([0 2*pi -1 1])
```



Each subregion contains its own axes with characteristics you can control independently of the other subregions. This example uses the `axis` function to set limits and change the shape of the subplots.

See the `axes`, `axis`, and `subplot` functions for more information.

Specifying the Target Axes

The current axes is the last one defined by `subplot`. If you want to access a previously defined subplot, for example to add a title, you must first make that axes current.

You can make an axes current in three ways:

- Click on the subplot with the mouse.
- Call `subplot` the `m`, `n`, `i` specifiers.
- Call `subplot` with the handle (identifier) of the axes.

For example,

```
subplot(2,2,2)
title('Top Right Plot')
```

adds a title to the plot in the upper right side of the figure.

You can obtain the handles of all the subplot axes with the statement

```
h = get(gcf,'Children');
```

The handles of all the axes are returned, with the most recently created one first. That is, `h(1)` is subplot 224, `h(2)` is subplot 223, `h(3)` is subplot 222, and `h(4)` is subplot 221. For example, to replace subplot 222 with a new plot, first make it the current axes with

```
subplot(h(3))
```

Default Color Scheme

The default figure color scheme produces good contrast and visibility for the various graphics functions. This scheme defines colors for the window background, the axis background, the axis lines and labels, the colors of the lines used for plotting and surface edges, and other properties that affect appearance.

The `colordef` function enables you to select from predefined color schemes and to modify colors individually. `colordef` predefines three color schemes:

- `colordef white` — Sets the axis background color to white, the window background color to gray, the colormap to `jet`, surface edge colors to black, and defines appropriate values for the plotting color order and other properties.
- `colordef black` — Sets the axis background color to black, the window background color to dark gray, the colormap to `jet`, surface edge colors to black, and defines appropriate values for the plotting color order and other properties.
- `colordef none` — Set the colors to match that of MATLAB Version 4. This is basically a black background with white axis lines and no grid. MATLAB programs that are based on the Version 4 color scheme may need to call `colordef` with the `none` option to produce the expected results.

You can examine the `colordef.m` M-file to determine what properties it sets (enter `type colordef` at the MATLAB prompt).

Using High-Level Plotting Functions

In this section...

Functions for Plotting Line Graphs

Many types of MATLAB functions are available for displaying vector data as line plots, as well as functions for annotating and printing these graphs. The following table summarizes the functions that produce basic line plots. These functions differ in the way they scale the plot's axes. Each accepts input in the form of vectors or matrices and automatically scales the axes to accommodate the data.

Function	Description
`plot`	Graph 2-D data with linear scales for both axes
`plot3`	Graph 3-D data with linear scales for both axes
`loglog`	Graph with logarithmic scales for both axes
`semilogx`	Graph with a logarithmic scale for the *x*-axis and a linear scale for the *y*-axis

Function	Description
`semilogy`	Graph with a logarithmic scale for the *y*-axis and a linear scale for the *x*-axis
`plotyy`	Graph with *y*-tick labels on the left and right side

To view a gallery of all the high level plot functions, also with links to their reference pages, see "Types of MATLAB Plots" on page 1-6.

Programmatic Plotting

The process of constructing a basic graph to meet your presentation graphics requirements is outlined in the following table. The table shows seven typical steps and some example code for each.

If you are performing analysis only, you may want to view various graphs just to explore your data. In this case, steps 1 and 3 may be all you need. If you are creating presentation graphics, you may want to fine-tune your graph by positioning it on the page, setting line styles and colors, adding annotations, and making other such improvements.

Step	Typical Code

1 Prepare your data	`x = 0:0.2:12;`
	`y1 = bessel(1,x);`
	`y2 = bessel(2,x);`
	`y3 = bessel(3,x);`

2 Select a window and position a plot region within the window	`figure(1)` `subplot(2,2,1)`
3 Call elementary plotting function	`h = plot(x,y1,x,y2,x,y3);`

4 Select line and marker characteristics	`set(h,'LineWidth',2,{'LineStyle'},`
	`{'--';':';'-.'})`
	`set(h,{'Color'},{'r';'g';'b'})`

Step	Typical Code
5 Set axis limits, tick marks, and grid lines	`axis([0 12 -0.5 1])` `grid on`

6 Annotate the graph with axis labels, legend, and text	`xlabel('Time')`
	`ylabel('Amplitude')`
	`legend(h,'First','Second','Third')`
	`title('Bessel Functions')`
	`[y,ix] = min(y1);`
	`text(x(ix),y,'First Min \rightarrow',...`
	`'HorizontalAlignment','right')`

7 Export graph	`print -depsc -tiff -r200 myplot`

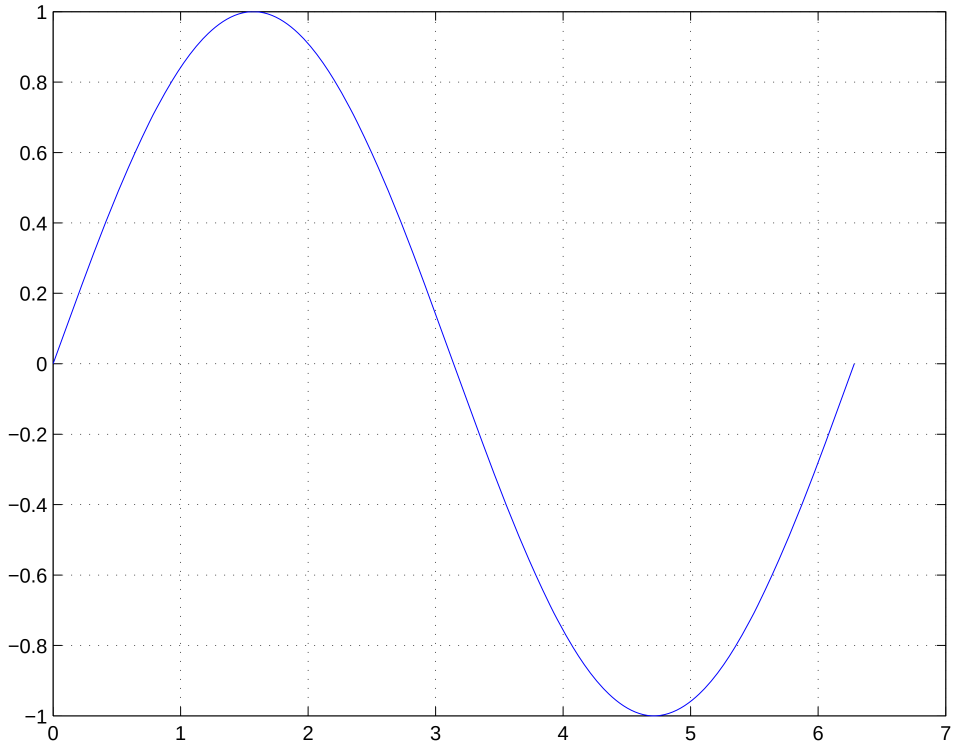
Creating Line Plots

The `plot` function has different forms depending on the input arguments. For example, if `y` is a vector, `plot(y)` produces a linear graph of the elements of `y` versus the index of the elements of `y`. If you specify two vectors as arguments, `plot(x,y)` produces a graph of `y` versus `x`.

For example, the following statements create a vector of values in the range [0, 2π] in increments of π/100 and then use this vector to evaluate the sine function over that range. MATLAB plots the vector on the *x*-axis and the value of the sine function on the *y*-axis.

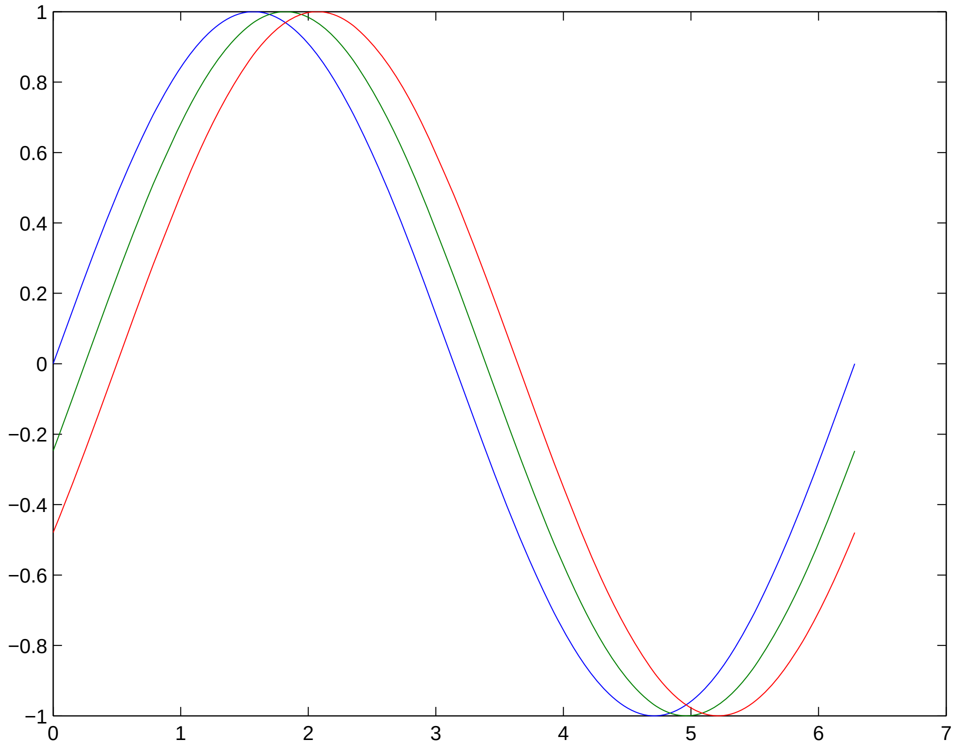
```
t = 0:pi/100:2*pi;
y = sin(t);
plot(t,y)
grid on % Turn on grid lines for this plot
```

Appropriate axis ranges and tick mark locations are automatically selected.



You can plot multiple graphs in one call to `plot` using *x-y* pairs. MATLAB automatically cycles through a predefined list of colors (determined by the axes `ColorOrder` property) to allow discrimination between sets of data. Plotting three curves as a function of `t` produces

```
y = sin(t);
y2 = sin(t-0.25);
y3 = sin(t-0.5);
plot(t,y,t,y2,t,y3)
```

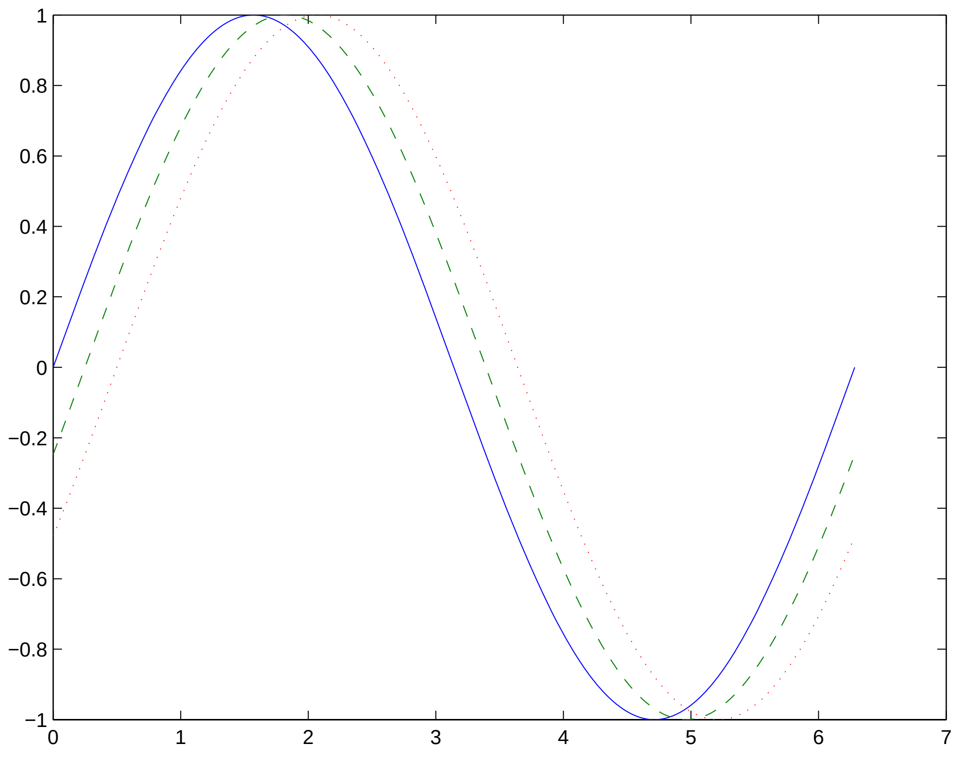


Specifying Line Style

You can assign different line styles to each data set by passing line style identifier strings to `plot`. For example,

```
t = 0:pi/100:2*pi;
y = sin(t);
y2 = sin(t-0.25);
y3 = sin(t-0.5);
plot(t,y,'-',t,y2,'--',t,y3,':')
```

The graph shows three lines of different colors and lines styles representing the value of the sine function with a small phase shift between each line, as defined by `y`, `y2`, and `y3`. The lines are blue solid, green dashed, and red dotted.



Colors, Line Styles, and Markers

The basic plotting functions accepts character-string arguments that specify various line styles, marker symbols, and colors for each vector plotted. In the general form,

```
plot(x,y,'linestyle_marker_color')
```

`linestyle_marker_color` is a character string (delineated by single quotation marks) constructed from

- A line style (e.g., dashed, dotted, etc.)
- A marker type (e.g., `x`, `*`, `o`, etc.)
- A predefined color specifier (`c`, `m`, `y`, `k`, `r`, `g`, `b`, `w`)

For example,

```
plot(x,y,':squarey')
```

plots a yellow dotted line and places square markers at each data point. If you specify a marker type, but not a line style, only the marker is plotted.

The specification can consist of one or none of each specifier in any order. For example, the string

```
'go--'
```

defines a dashed line with circular markers, both colored green.

You can also specify the size of the marker and, for markers that are closed shapes, you can specify separately the colors of the edges and the face.

See the `LineSpec` discussion for more information.

Specifying the Color and Size of Lines

You can control a number of line style characteristics by specifying values for line properties:

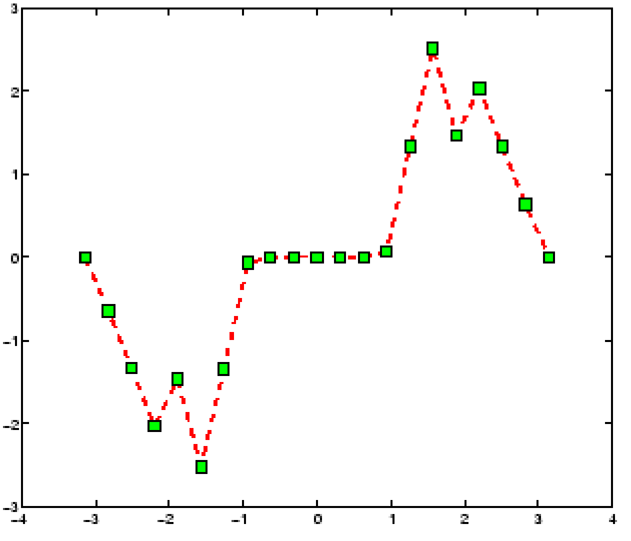
- `LineWidth` — Width of the line in units of points
- `MarkerEdgeColor` — Color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles)
- `MarkerFaceColor` — Color of the face of filled markers
- `MarkerSize` — Size of the marker in units of points

For example, these statements,

```
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
plot(x,y,'--rs','LineWidth',2,...
                'MarkerEdgeColor','k',...
                'MarkerFaceColor','g',...
                'MarkerSize',10)
```

produce a graph with

- A red dashed line with square markers
- A line width of two points
- The edge of the marker colored black
- The face of the marker colored green
- The size of the marker set to 10 points



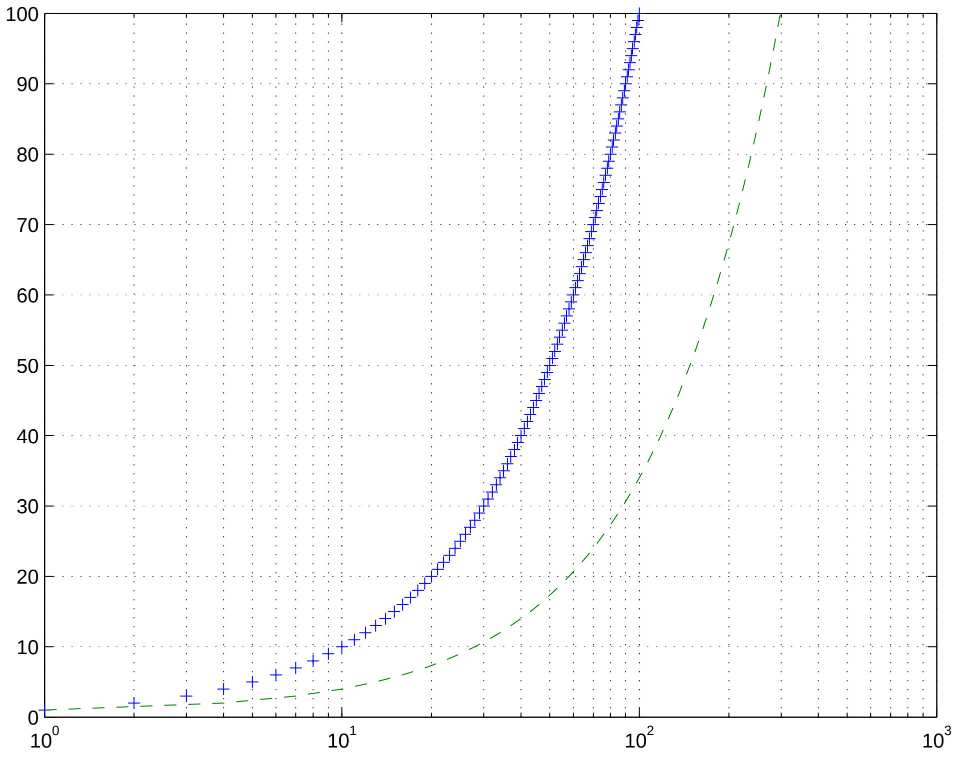
Adding Plots to an Existing Graph

You can add plots to an existing graph using the `hold` command. When you set `hold` to `on`, MATLAB does not remove the existing graph; it adds the new data to the current graph, rescaling if the new data falls outside the range of the previous axis limits.

For example, these statements first create a semilogarithmic plot, then add a linear plot.

```
semilogx(1:100,'+')
hold all % hold plot and cycle line colors
plot(1:3:300,1:100,'--')
hold off
grid on % Turn on grid lines for this plot
```

The *x*-axis limits are rest to accommodate the new data, but the scaling from logarithmic to linear does not change.



Plotting Only the Data Points

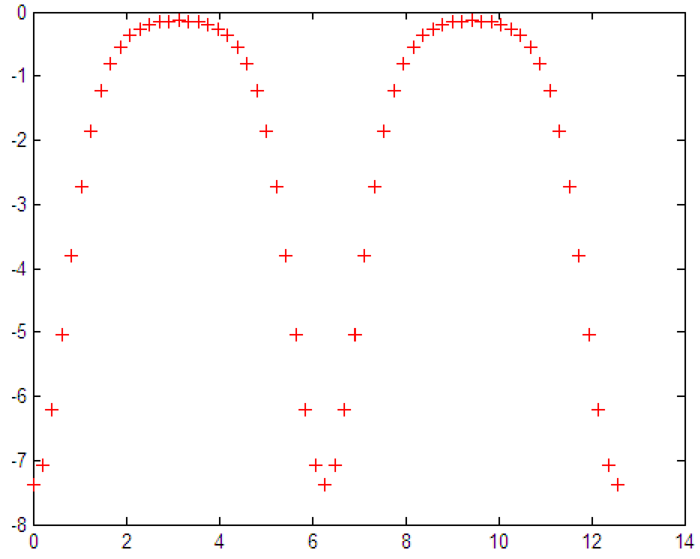
To plot a marker at each data point without connecting the markers with lines, use a specification that does not contain a line style. For example, given two vectors,

```
x = 0:pi/15:4*pi;
y = -exp(2*cos(x));
```

calling `plot` with only a color and marker specifier

```
plot(x,y,'r+')
```

plots a red plus sign at each data point.

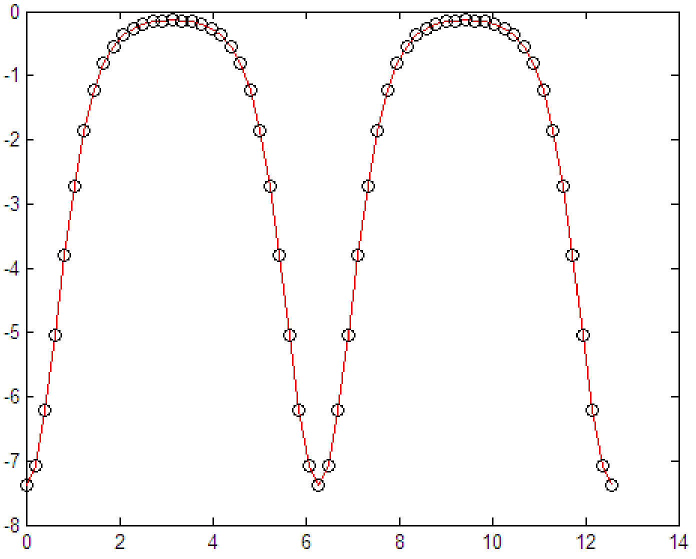


See `LineSpec` for a list of available line styles, markers, and colors.

Plotting Markers and Lines

To plot both markers and the lines that connect them, specify a line style and a marker type. For example, the following code plots the data as a red, solid line and then adds circular markers with black edges at each data point.

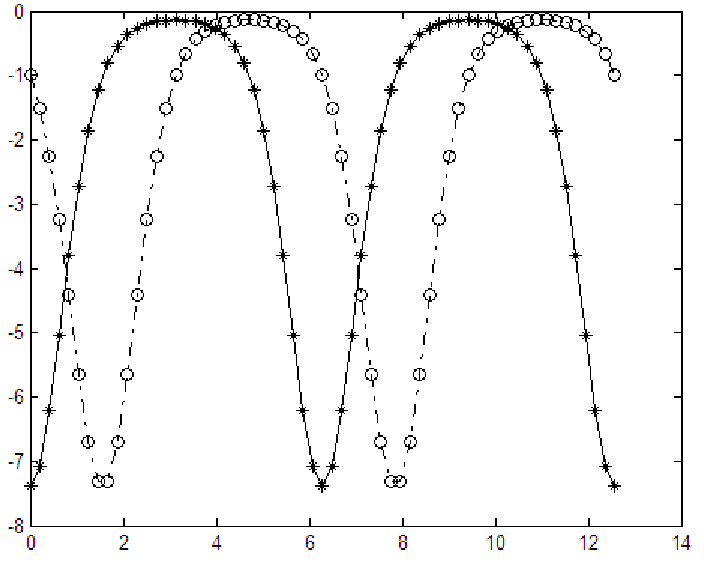
```
x = 0:pi/15:4*pi;
y = -exp(2*cos(x));
plot(x,y,'-r',x,y,'ok')
```



Line Styles for Black and White Output

Line styles and markers enable you to discriminate different plots on the same graph when color is not available. For example, the following statements create a graph using a solid ('-*k') line with asterisk markers colored black and a dash-dot ('-.ok') line with circular markers colored black.

```
x = 0:pi/15:4*pi;
y1 = -exp(2*cos(x));
y2 = -exp(2*sin(x));
plot(x,y1,'-*k',x,y2,'-.ok')
```



Setting Default Line Styles

You can configure MATLAB defaults to use line styles instead of colors for multiline plots by setting a value for the axes `LineStyleOrder` property using a cell array of linespecs. For example, the command

```
set(0,'DefaultAxesLineStyleOrder',{'-o',':s','--+'})
```

defines three line styles and makes them the default for all plots.

To set the default line color to dark gray, use the statement

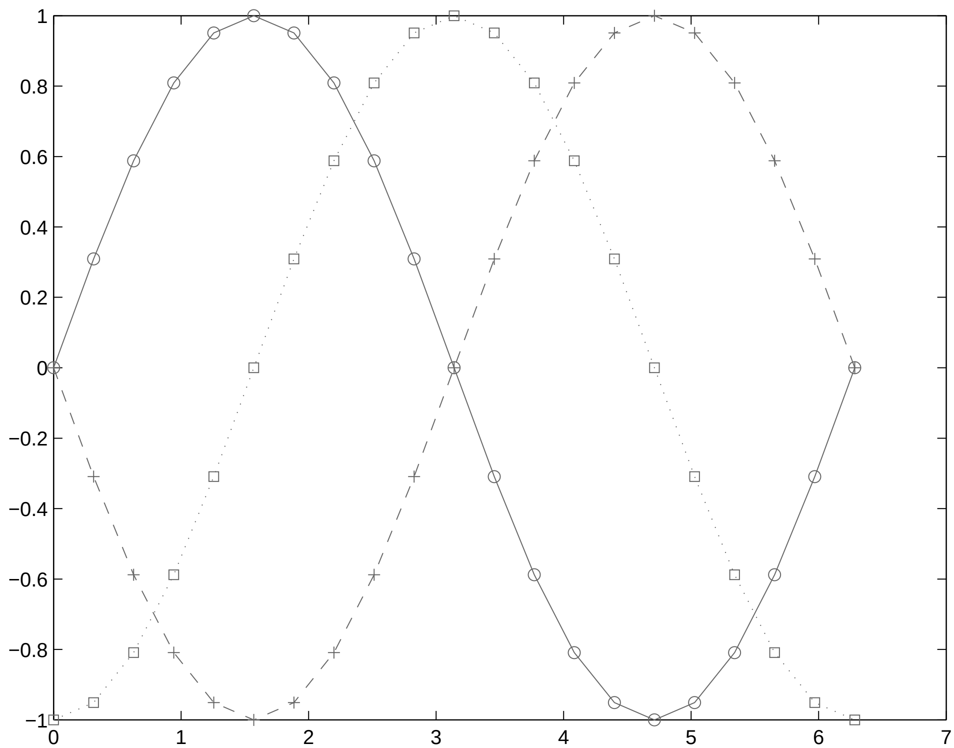
```
set(0,'DefaultAxesColorOrder',[0.4,0.4,0.4])
```

See `ColorSpec` for information on how to specify color as a three-element vector of RGB values.

Now the plot function uses the line styles and colors you have defined as defaults. For example, these statements create a multiline plot.

```
x  = 0:pi/10:2*pi;
y1 = sin(x);
```

```
y2 = sin(x-pi/2);
y3 = sin(x-pi);
plot(x,y1,x,y2,x,y3)
```



The default values persist until you quit MATLAB. To remove default values during your MATLAB session, use the reserved word `remove`.

```
set(0,'DefaultAxesLineStyleOrder','remove')
set(0,'DefaultAxesColorOrder','remove')
```

See "Setting Default Property Values" on page 8-52 for more information.

Line Plots of Matrix Data

When you call the plot function with a single matrix argument

```
plot(Y)
```

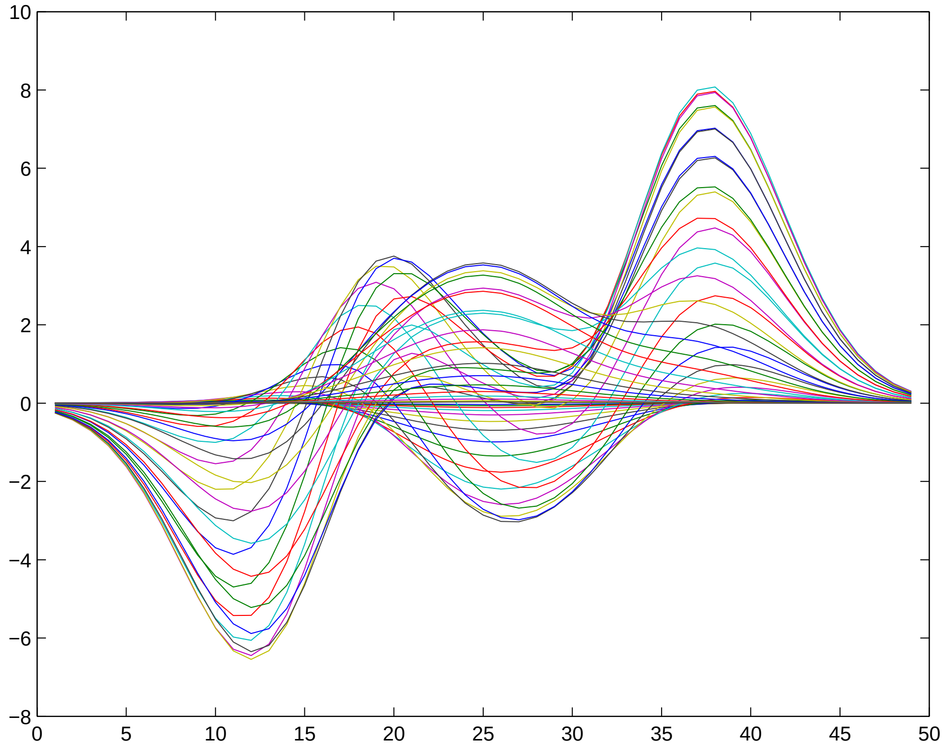
One line is plotted for each column of the matrix. The *x*-axis is labeled with the row index vector 1:m, where m is the number of rows in Y. For example,

```
Z = peaks;
```

returns a 49-by-49 matrix obtained by evaluating a function of two variables. Plotting this matrix

```
plot(Z)
```

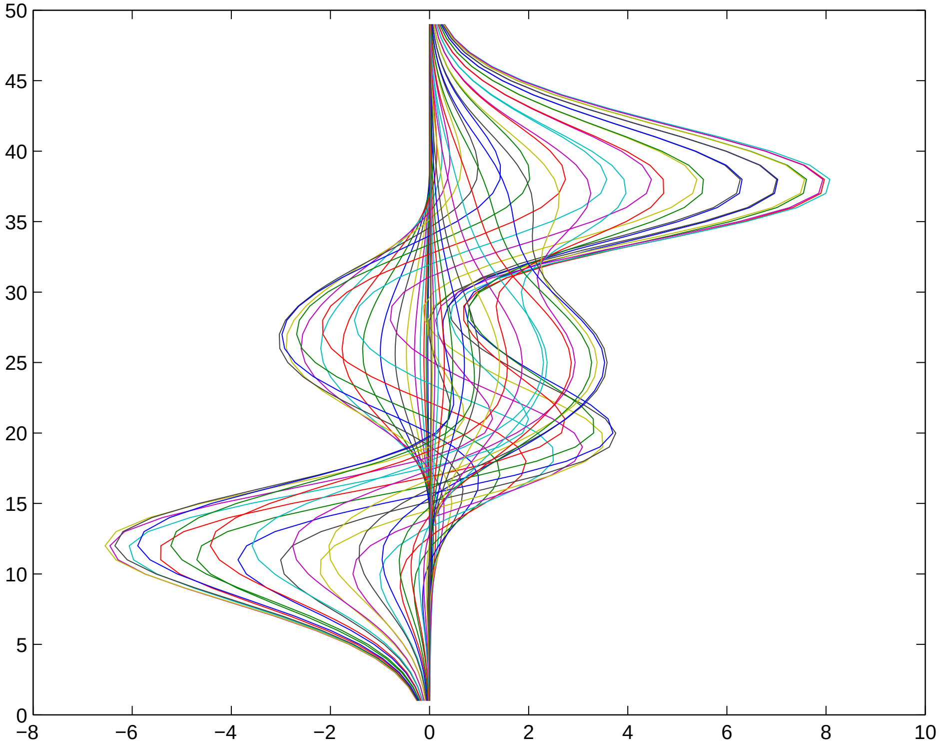
produces a graph with 49 lines.



In general, if `plot` is used with two arguments and if either `X` or `Y` has more than one row or column, then

- If `Y` is a matrix, and `x` is a vector, `plot(x,Y)` successively plots the rows or columns of `Y` versus vector `x`, using different colors or line types for each. The row or column orientation varies depending on whether the number of elements in `x` matches the number of rows in `Y` or the number of columns. If `Y` is square, its columns are used.
- If `X` is a matrix and `y` is a vector, `plot(X,y)` plots each row or column of `X` versus vector `y`. For example, plotting the `peaks` matrix versus the vector `1:length(peaks)` rotates the previous plot.

```
y = 1:length(peaks);
plot(peaks,y)
```



- If X and Y are both matrices of the same size, plot(X,Y) plots the columns of X versus the columns of Y.

You can also use the plot function with multiple pairs of matrix arguments.

```
plot(X1,Y1,X2,Y2,...)
```

This statement graphs each X-Y pair, generating multiple lines. The different pairs can be of different dimensions.

Plotting Imaginary and Complex Data

When the arguments to `plot` are complex (i.e., the imaginary part is nonzero), All MATLAB graphics functions ignore the imaginary part *except* when `plot` is given a single complex data argument. For this special case, the command produces a plot of the real part versus the imaginary part. Therefore,

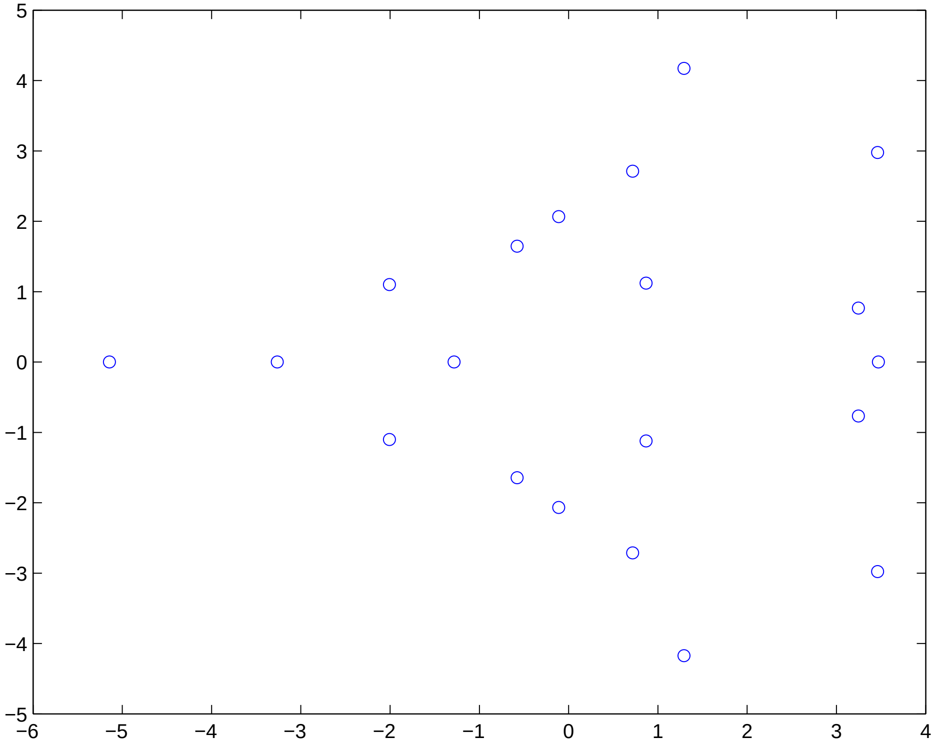
```
plot(Z)
```

where `Z` is a complex vector or matrix, is equivalent to

```
plot(real(Z),imag(Z))
```

For example, this statement plots the distribution of the eigenvalues of a random matrix using circular markers to indicate the data points.

```
plot(eig(randn(20,20)),'o','MarkerSize',6)
```



To plot more than one complex matrix, there is no shortcut; the real and imaginary parts must be taken explicitly.

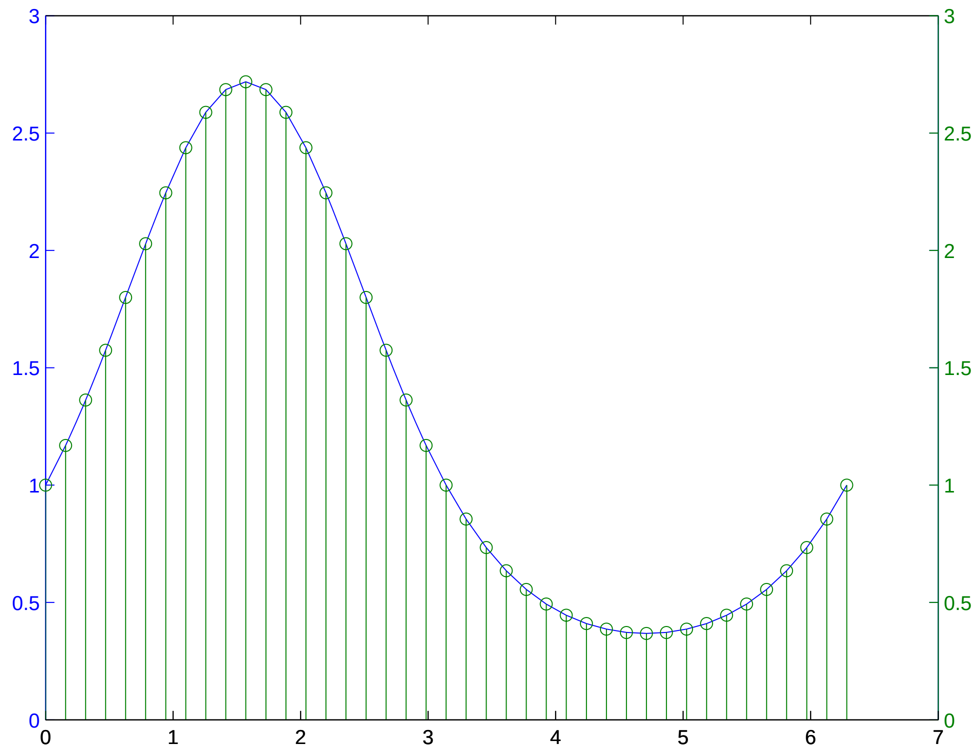
Plotting with Two Y-Axes

In this section...

Introduction

The `plotyy` function enables you to create plots of two data sets and use both left and right side *y*-axes. You can also apply different plotting functions to each data set. For example, you can combine a line plot with a stem plot of the same data.

```
t = 0:pi/20:2*pi;
y = exp(sin(t));
plotyy(t,y,t,y,'plot','stem')
```



Combining Linear and Logarithmic Axes

You can use `plotyy` to apply linear and logarithmic scaling to compare two data sets having different ranges of values.

```
t = 0:900; A = 1000; a = 0.005; b = 0.005;
z1 = A*exp(-a*t);
z2 = sin(b*t);
[haxes,hline1,hline2] = plotyy(t,z1,t,z2,'semilogy','plot');
```

This example saves the handles of the lines and axes created to adjust and label the graph. First, label the axes whose *y* value ranges from 10 to 1000.

This is the first handle in `haxes` because it was specified first in the call to `plotyy`. Use the `axes` function to make `haxes(1)` the current axes, which is then the target for the `ylabel` function.

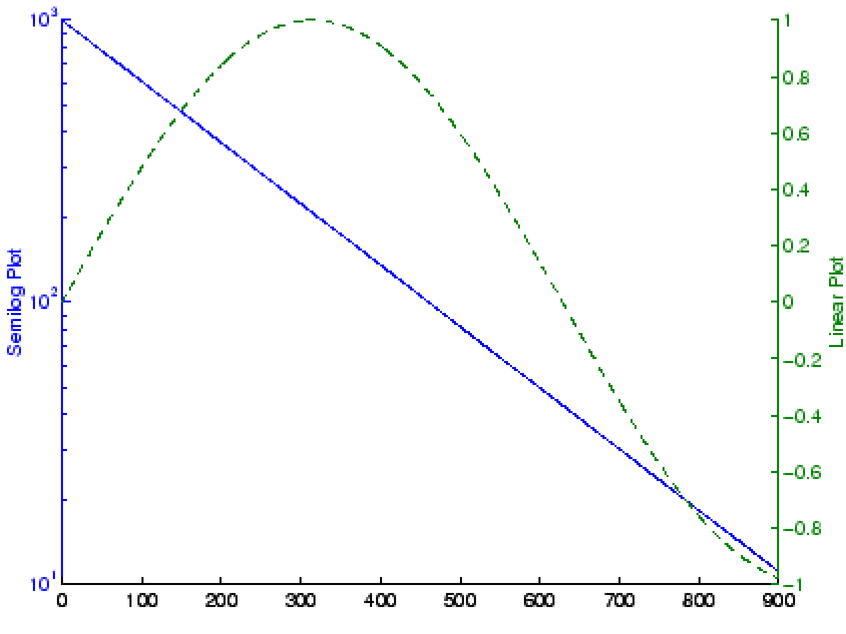
```
axes(haxes(1))
ylabel('Semilog Plot')
```

Now make the second axes current and call `ylabel` again.

```
axes(haxes(2))
ylabel('Linear Plot')
```

You can modify the characteristics of the plotted lines in a similar way. For example, to change the line style of the second line plotted to a dashed line, use the statement

```
set(hline2,'LineStyle','--')
```



See "Using Multiple X- and Y-Axes" on page 10-25 for an example that employs double *x*- and *y*-axes.

See `LineSpec` for additional line properties.

Setting Axis Parameters

In this section...

Axis Scaling and Ticks

When you create a MATLAB graph, the axis limits and tick-mark spacing are automatically selected based on the data plotted. However, you can also specify your own values for axis limits and tick marks with the following functions:

- `axis` — Sets values that affect the current axes object (the most recently created or the last clicked on).
- `axes` — (Not axis) creates a new axes object with the specified characteristics.
- `get` and `set` — Enable you to query and set a wide variety of properties of existing axes.
- `gca` — Returns the handle (identifier) of the current axes. If there are multiple axes in the figure window, the current axes is the last graph created or the last graph you clicked on with the mouse. The following two sections provide more information and examples:

See "Defining the View" in the 3-D Visualization documentation for more extensive information on manipulating 3-D views.

Axis Limits and Ticks

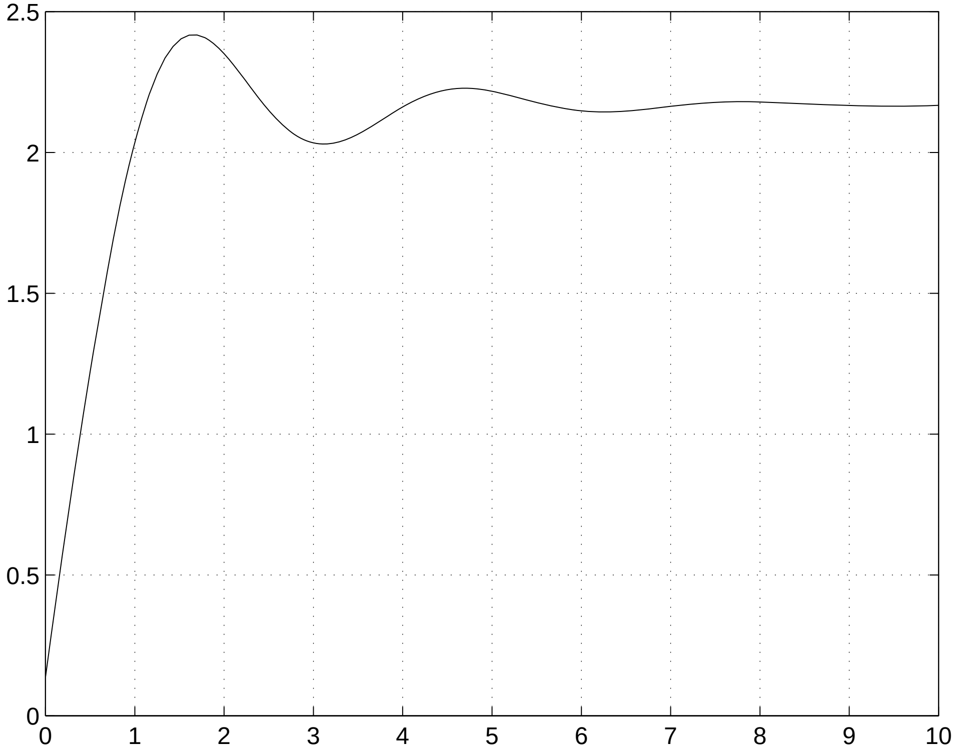
By default, axis limits are chosen to encompass the range of the plotted data. You can specify the limits manually using the `axis` function. Call `axis` with the new limits defined as a four-element vector.

```
axis([xmin,xmax,ymin,ymax])
```

The minimum values must be less than the maximum values.

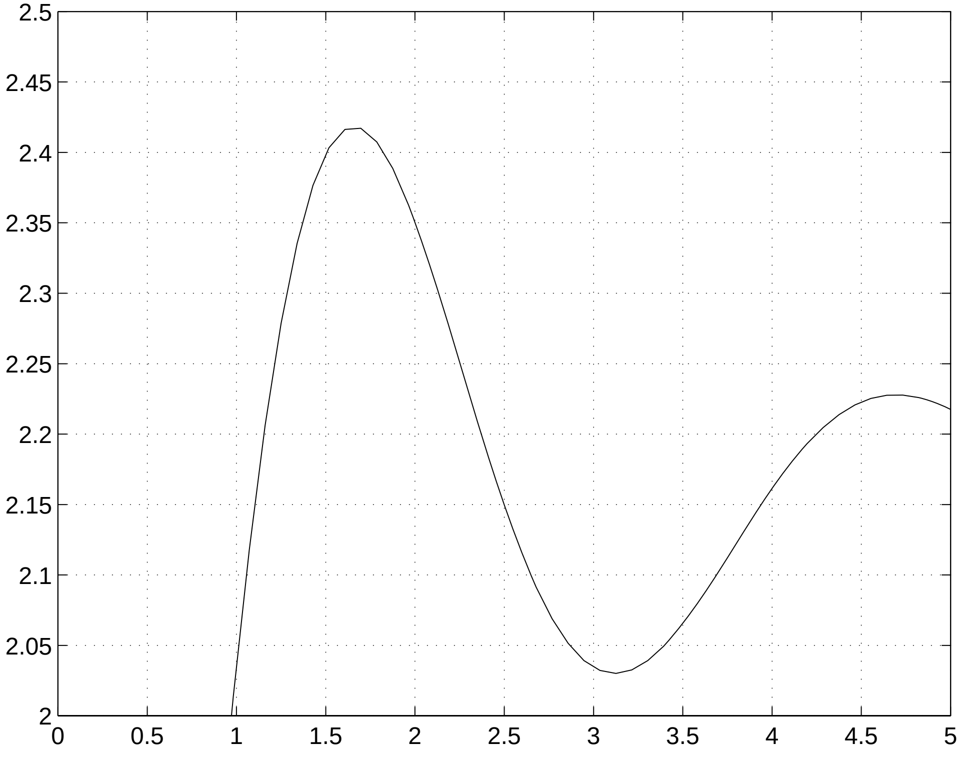
Semiautomatic Limits

If you want to autoscale only one of a min/max set of axis limits, but you want to specify the other, use the MATLAB variable `Inf` or `-Inf` for the autoscaled limit. For example, this graph uses default scaling.



Compare the default limits to the following graph, which sets the maximum limit of the *x*-axis, but autoscales the minimum limit.

```
axis([-Inf 5 2 2.5])
```



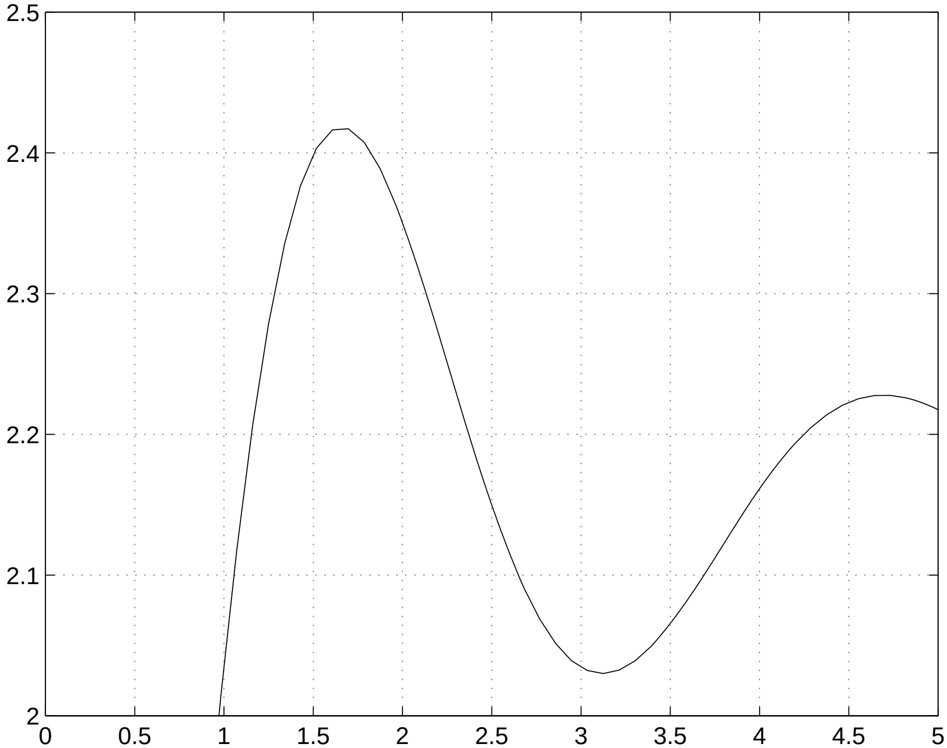
Axis Tick Marks

The tick-mark locations are based on the range of data so as to produce equally spaced ticks (for linear graphs). You can specify different tick marks by setting the axes `XTick` and `YTick` properties. Define tick marks as a vector of increasing values. The values do not need to be equally spaced.

For example, setting the *y*-axis tick marks for the graph from the preceding example,

```
set(gca,'ytick',[2 2.1 2.2 2.3 2.4 2.5])
```

produces a graph with only the specified ticks on the *y*-axis.



If you specify tick-mark values that are outside the axis limits, they are not displayed (that is, specifying tick marks cannot cause axis limits to change).

Example — Specifying Ticks and Tick Labels

You can adjust the axis tick-mark locations and the labels appearing at each tick mark. For example, this plot of the sine function relabels the *x*-axis with more meaningful values.

```
x = -pi:.1:pi;
y = sin(x);
plot(x,y)
set(gca,'XTick',-pi:pi/2:pi)
set(gca,'XTickLabel',{'-pi','-pi/2','0','pi/2','pi'})
```

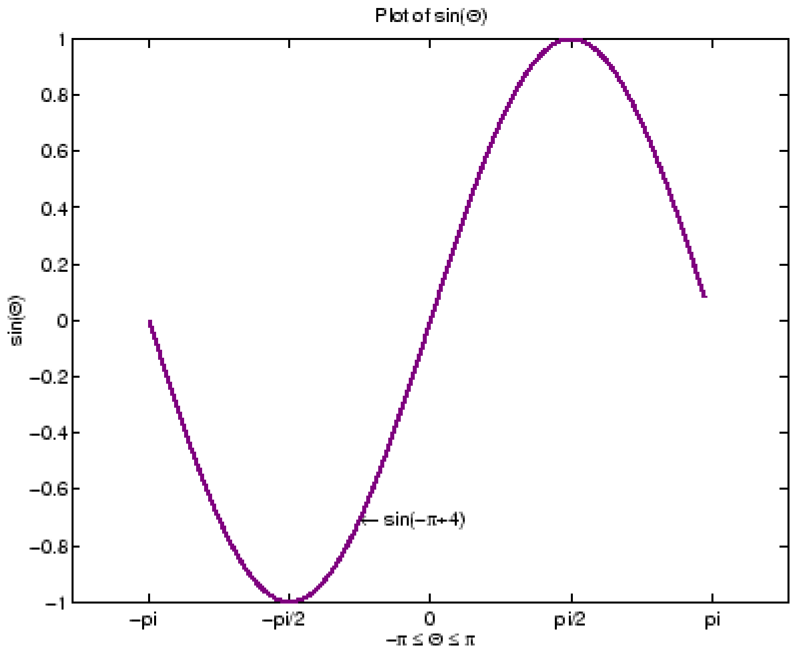
These functions (`xlabel`, `ylabel`, `title`, `text`) add axis labels and draw an arrow that points to the location on the graph where y = sin(-pi/4).

```
xlabel('-\pi \leq \Theta \leq \pi')
ylabel('sin(\Theta)')
title('Plot of sin(\Theta)')
text(-pi/4,sin(-pi/4),'\leftarrow sin(-\pi\div4)',...
     'HorizontalAlignment','left')
```

Setting Line Properties on an Existing Plot

Change the line color to purple by first finding the handle of the line object created by `plot` and then setting its `Color` property. Use `findobj` and the fact that the MATLAB line color default is a blue line (RGB value [0 0 1]). In the same statement, set the `LineWidth` property to 2 points.

```
set(findobj(gca,'Type','line','Color',[0 0 1]),...
    'Color',[0.5,0,0.5],'LineWidth',2)
```



The Greek symbols are created using TeX character sequences.

Setting Aspect Ratio

By default, graphs display in a rectangular axes that has the same aspect ratio as the figure window. This makes optimum use of space available for plotting. You exercise control over the aspect ratio with the `axis` function.

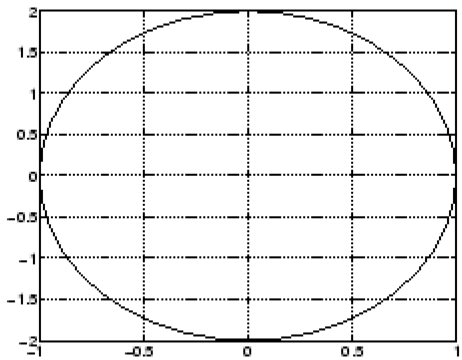
For example,

```
t = 0:pi/20:2*pi;
plot(sin(t),2*cos(t))
grid on
```

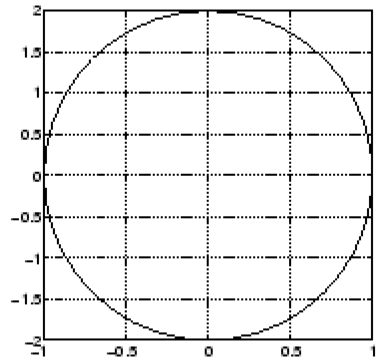
produces a graph with the default aspect ratio. The command

```
axis square
```

makes the *x*- and *y*-axes equal in length.



`axis normal`

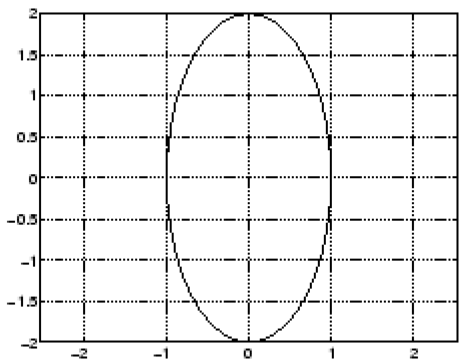


`axis square`

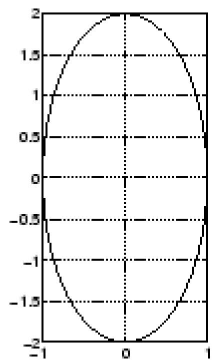
The square axes has one data unit in *x* to equal two data units in *y*. If you want the *x*- and *y*-data units to be equal, use the command

```
axis equal
```

This produces an axes that is rectangular in shape, but has equal scaling along each axis.



`axis equal`



`axis equal tight`

If you want the axes shape to conform to the plotted data, use the `tight` option in conjunction with `equal`.

```
axis equal tight
```

Note In order to format aspect ratio using `axis`, axes must exist and contain a plot. That is, you cannot pre-format an axes that has no actual *x*-, *y*-, or *z*-limits. To overcome this, you can preformat the axes with `axis` and issue the `hold on` command before plotting data.

5

Creating Specialized Plots

Bar and Area Graphs

In this section...

Types of Bar Graphs

Bar and area graphs display vector or matrix data. These types of graphs are useful for viewing results over a period of time, comparing results from different data sets, and showing how individual elements contribute to an aggregate amount. Bar graphs are suitable for displaying discrete data, whereas area graphs—like line graphs—are more suitable for displaying continuous data. This table lists the functions that plot bar and area graphs.

Function	Description
`bar`	Displays columns of m-by-n matrix as m groups of n vertical bars.
`barh`	Displays columns of m-by-n matrix as m groups of n horizontal bars.
`bar3`	Displays columns of m-by-n matrix as m groups of n vertical 3-D bars.
`bar3h`	Displays columns of m-by-n matrix as m groups of n horizontal 3-D bars.
`area`	Displays vector data as stacked area plots.

Four of these five functions display bar graphs (there is only one type of area graph; see "Area Graphs" on page 5-20). Bar graphs differ according to whether they plot in 2-D or 3-D and create vertical or horizontal bars, as this table describes.

Orientation	Two-Dimensional	Three-Dimensional
Vertical	`bar`	`bar3`
Horizontal	`barh`	`bar3h`

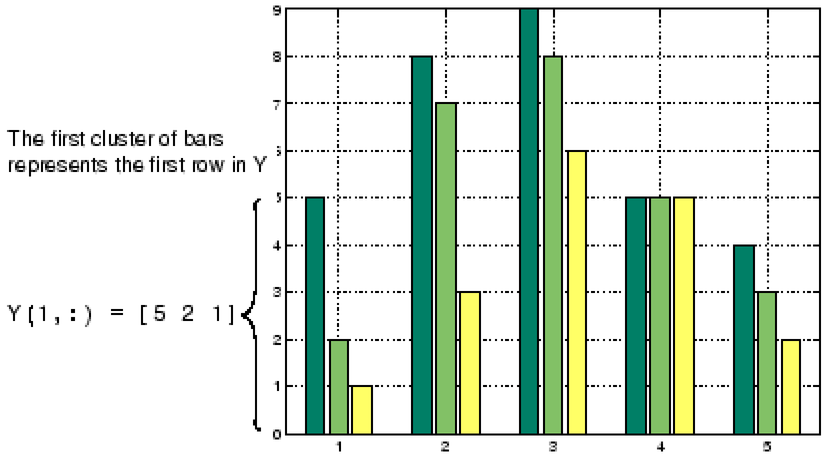
Grouped Bar Graph

By default, a bar graph represents each element in a matrix as one bar. Bars in a 2-D bar graph, created by the `bar` function, are distributed along the *x*-axis, with each element in a column drawn at a different location. All elements in a row are clustered around the same location on the *x*-axis.

For example, define `Y` as a simple matrix and issue the `bar` function in its simplest form:

```
Y = [5 2 1
     8 7 3
     9 8 6
     5 5 5
     4 3 2];
bar(Y)
```

The bars are clustered together by rows and evenly distributed along the *x*-axis.

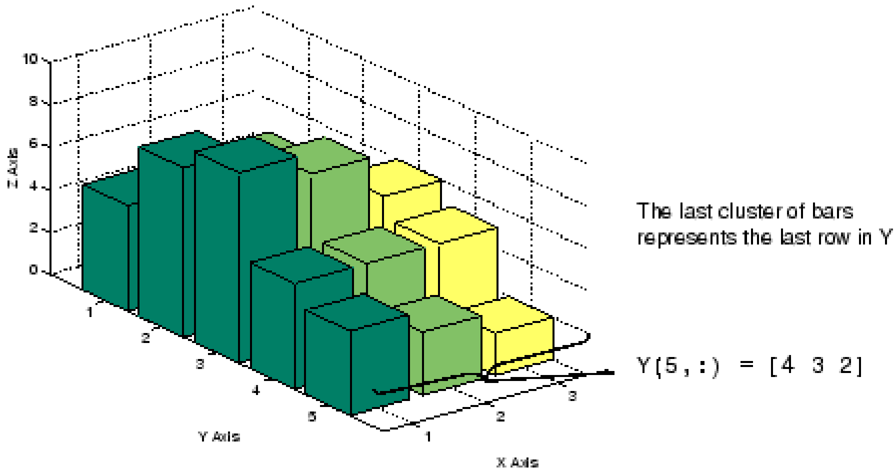


Detached 3-D Bars

The `bar3` function, in its simplest form, draws each element as a separate 3-D block, with the elements of each column distributed along the *y*-axis. Bars that represent elements in the first column of the matrix are centered at `1` along the *x*-axis. Bars that represent elements in the last column of the matrix are centered at `size(Y,2)` along the *x*-axis. For example,

```
bar3(Y)
```

displays five groups of three bars along the *y*-axis. Notice that larger bars obscure `Y(1,2)` and `Y(1,3)`.



By default, `bar3` draws detached bars. The statement `bar3(Y,'detach')` has the same effect.

Labeling the Graph. To add axes labels and *x* tick marks to this bar graph, use these statements:

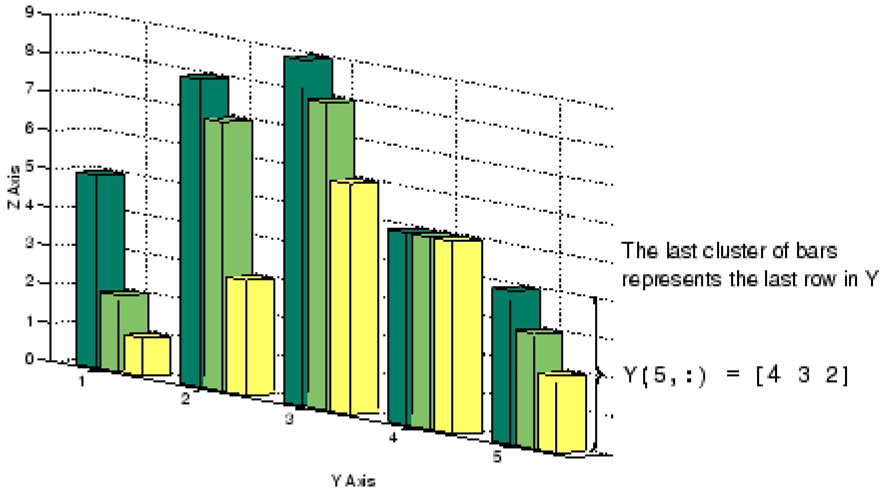
```
xlabel('X Axis')
ylabel('Y Axis')
zlabel('Z Axis')
set(gca,'XTick',[1 2 3])
```

Grouped 3-D Bars

Cluster the bars from each row beside each other by specifying the argument `'group'`. For example:

```
bar3(Y,'group')
```

groups the bars according to row and distributes the clusters evenly along the *y*-axis.

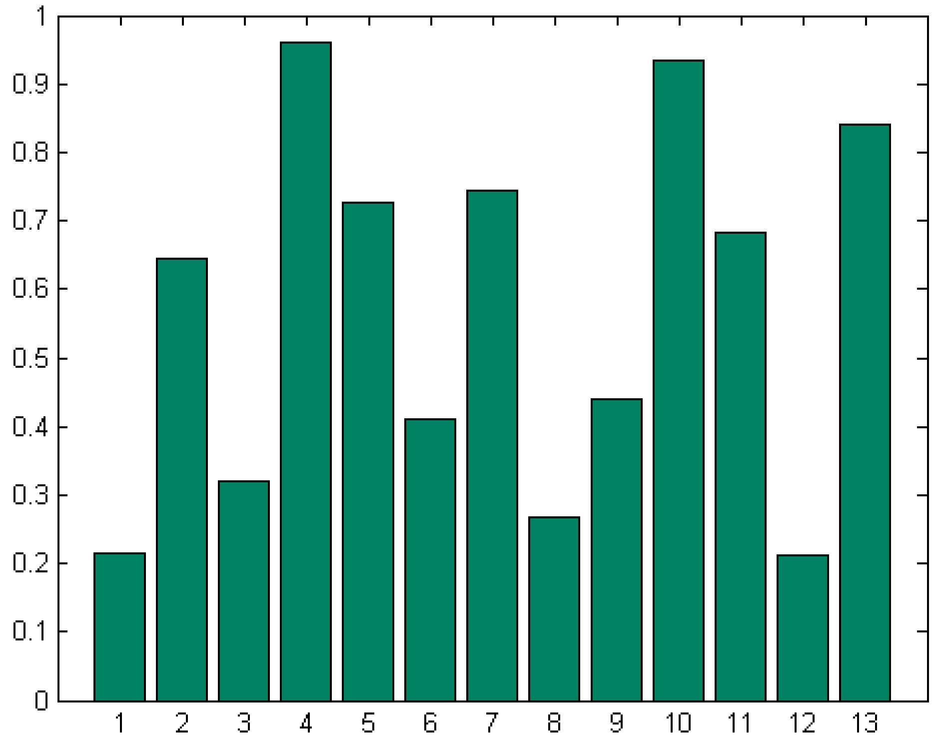


Coloring 2-D Bars According to Height

The `bar` and `barh` functions make all bars in a series the same color. With a little effort, however, you can assign a desired color to each bar. The typical approach is to associate bar colors with bar heights (*y* values). The following steps describe one way to do this, first using faceted shading and then using smooth (interpolated) shading:

1 Make up some numbers, plot a default bar plot, and assign a bichromatic colormap:

```
n = 13;
Z = rand(n,1);
h = bar(Z);
colormap(summer(n));
```



Only the first color is used to color the faces.

2 Assign a new color to each bar. `bar` (and `barh`) creates a barseries object, which encapsulates a set of patch objects for the bars. The patches have face-vertex syntax. First get a handle for the children, and then obtain the vertices for the bars and the vertex color data:

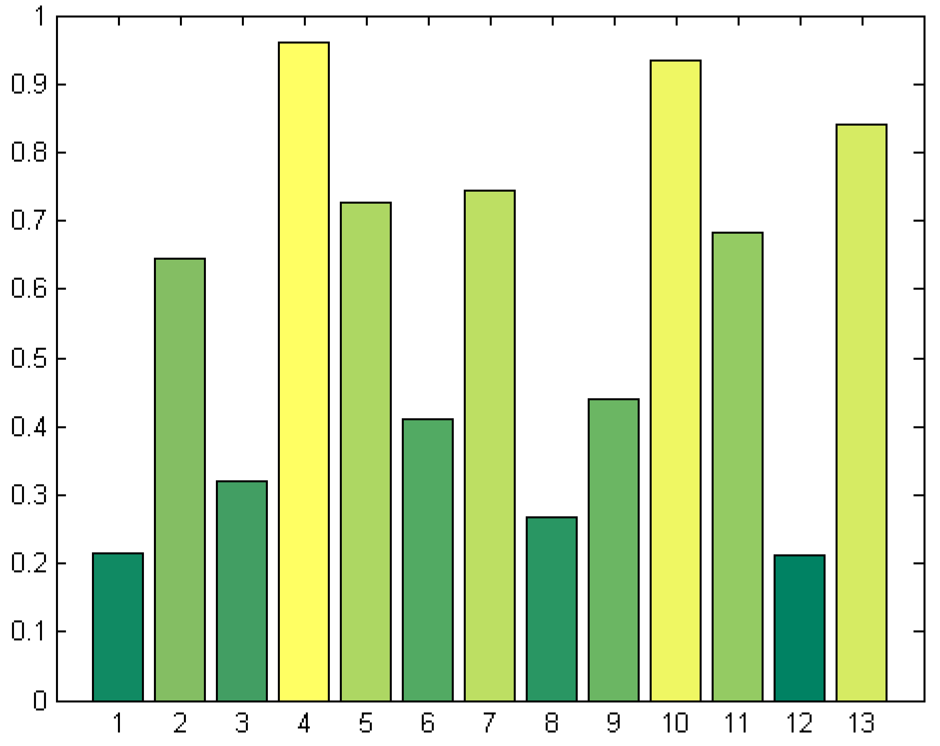
```
ch = get(h,'Children');
fvd = get(ch,'Faces');
fvcd = get(ch,'FaceVertexCData');
```

3 Sort the data to obtain an index for traversing the Faces array from the lowest to highest bar:

```
[zs, izs] = sortrows(Z,1);
```

4 Traverse the Faces array and assign colors to the face-vertex color data as you go:

```
for i = 1:n
    row = izs(i);
    fvcd(fvd(row,:)) = i;
end
set(ch,'FaceVertexCData',fvcd)
```

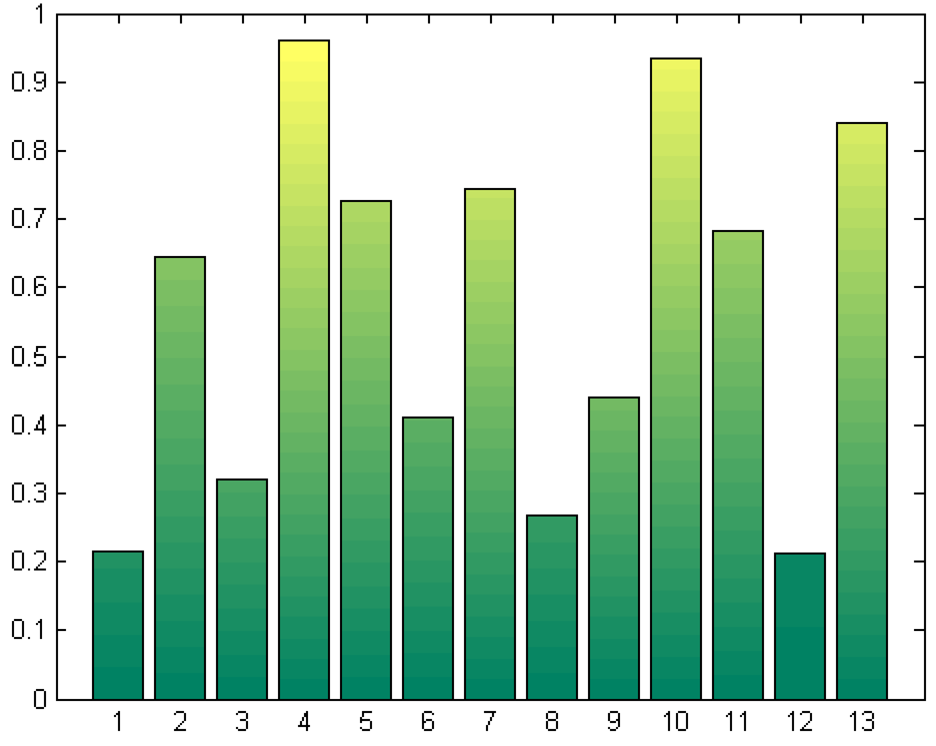


The code assigns colors to bars based on their `YData` ranks, rather than on their `YData` values. This helps to distinguish bars by color, but also the code can assign to bars that are nearly the same height a wider range of colors than if the colors were directly mapped to `YData` values.

5 To make the graph more readable, you can set different colors for vertices on the baseline and on the top, and then apply interpolated shading to

change hue going up the bars. The following code colors the two vertices at the base of each bar using the first color in the colormap, and assigns a color to the two vertices at the top proportionally to bar height. A longer color ramp than was used previously is needed to obtain smooth gradations of shading:

```
k = 128;                % Number of colors in color table
colormap(summer(k));    % Expand the previous colormap
shading interp          % Needed to graduate colors
for i = 1:n
    color = floor(k*i/n);       % Interpolate a color index
    row = izs(i);               % Look up actual row # in data
    fvcd(fvd(row,1)) = 1;       % Color base vertices 1st index
    fvcd(fvd(row,4)) = 1;
    fvcd(fvd(row,2)) = color;   % Assign top vertices color
    fvcd(fvd(row,3)) = color;
end
set(ch,'FaceVertexCData', fvcd);  % Apply the vertex coloring
set(ch,'EdgeColor','k')           % Give bars black borders
```



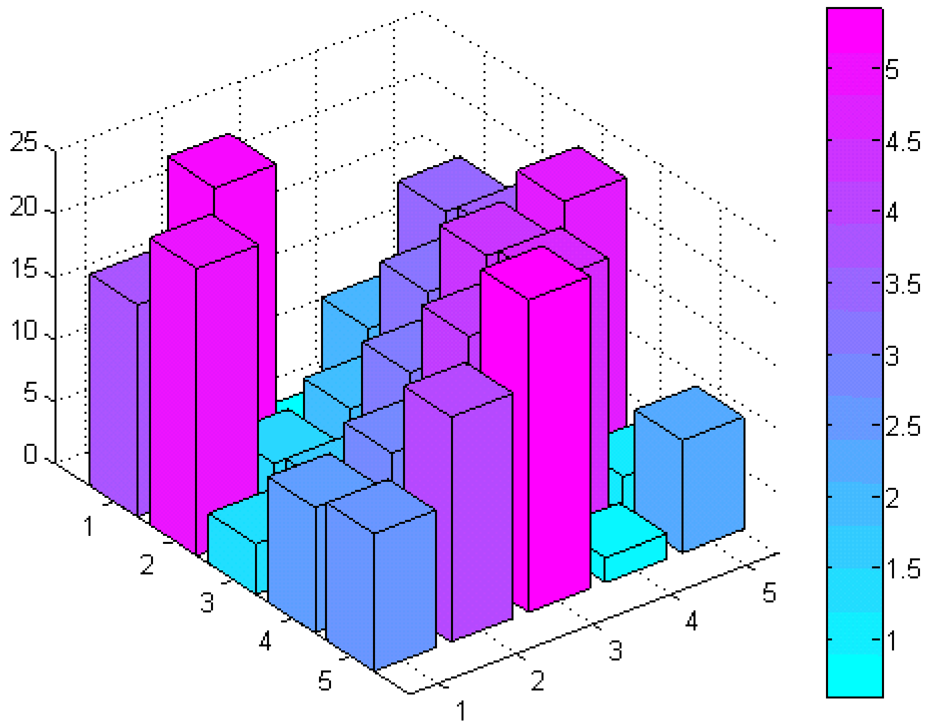
Coloring 3-D Bars According to Height

By default, all bars in a series (column) have the same color. You can modify a 3-D bar plot to color each bar according to how tall it is, but the technique is slightly different than the one used for coloring 2-D bars. Applying a monochromatic or bichromatic colormap to such plots helps viewers see height distinctions more readily. Adding a colorbar can also help.

The graph reads better if you override the default behavior of `bar3` to shade the sides of the bars with contrasting hues. You can color bars by height and make the sides match the color of the top of each bar by executing the following code:

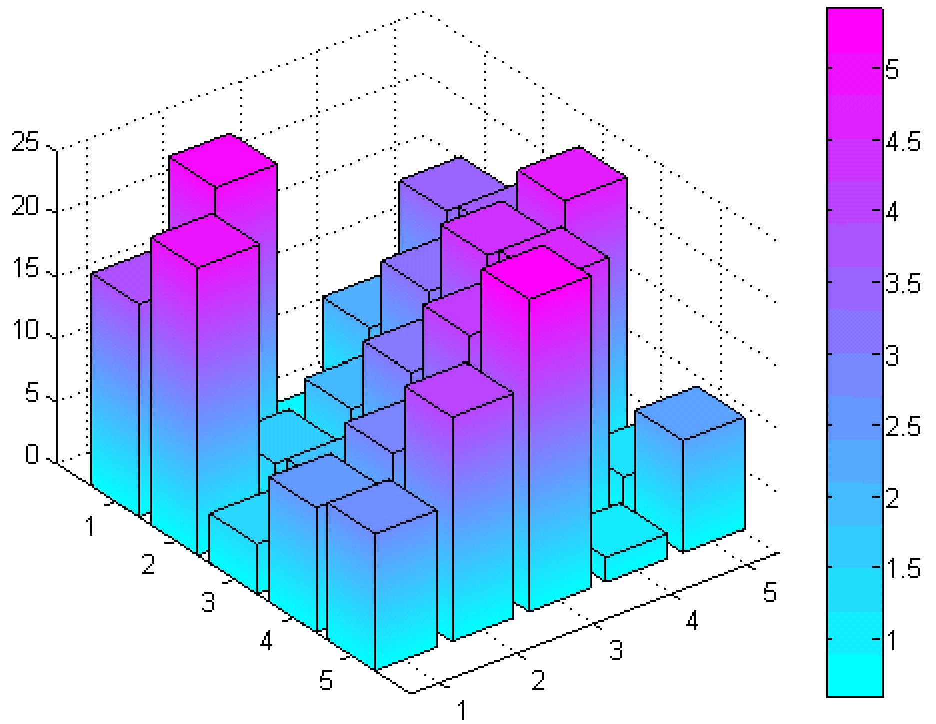
```
Z = magic(5);
h = bar3(Z);
```

```
for i = 1:length(h)
    zdata = ones(6*length(h),4);
    k = 1;
    for j = 0:6:(6*length(h)-6)
        zdata(j+1:j+6,:) = Z(k,i);
        k = k+1;
    end
    set(h(i),'Cdata',zdata)
end
colormap cool
colorbar
```



You can then make the plot even more readable by interpolating colors along the bars and giving their `EdgeColor` a contrasting color. The following code accomplishes this:

```
shading interp
for i = 1:length(h)
    zdata = get(h(i),'Zdata');
    set(h(i),'Cdata',zdata)
    set(h,'EdgeColor','k')
end
```



Stacked Bar Graphs to Show Contributing Amounts

Bar graphs can show how elements in the same row of a matrix contribute to the sum of all elements in the row. These types of bar graphs are referred to as stacked bar graphs.

Stacked bar graphs display one bar per row of a matrix. The bars are divided into *n* segments, where *n* is the number of columns in the matrix. For vertical

bar graphs, the height of each bar equals the sum of the elements in the row. Each segment is equal to the value of its respective element.

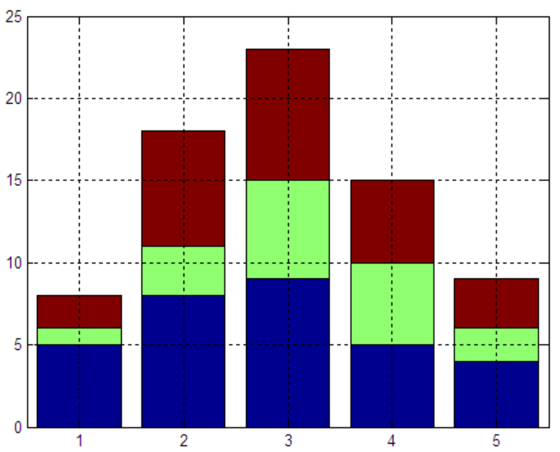
Redefining Y

```
Y = [5 1 2
     8 3 7
     9 6 8
     5 5 5
     4 2 3];
```

Create stacked bar graphs using the optional `'stack'` argument. For example:

```
bar(Y,'stack')
grid on
set(gca,'Layer','top') % display gridlines on top of graph
```

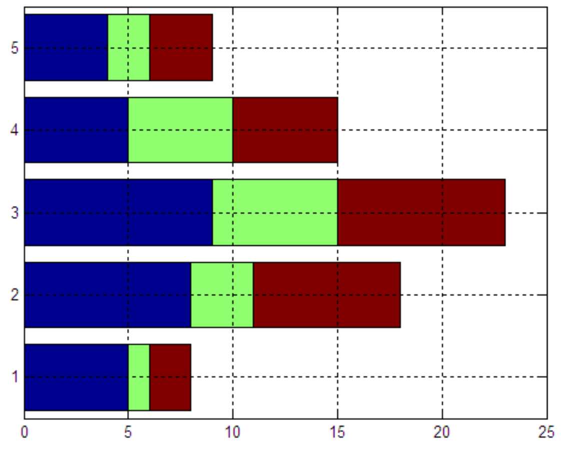
creates a 2-D stacked bar graph, where all elements in a row correspond to the same *x* location.



Horizontal Bar Graphs

For horizontal bar graphs, the length of each bar equals the sum of the elements in the row. The length of each segment is equal to the value of its respective element.

```
barh(Y,'stack')
grid on
set(gca,'Layer','top') % Display gridlines on top of graph
```



Specifying X-Axis Data

Bar graphs automatically generate *x*-axis values and label the *x*-axis tick lines. Specify a vector of *x* values (or *y* values in the case of horizontal bar graphs) to label the axes.

For example, given temperature data,

```
temp = [29 23 27 25 20 23 23 27];
```

obtained from samples taken every five days during a thirty-five day period,

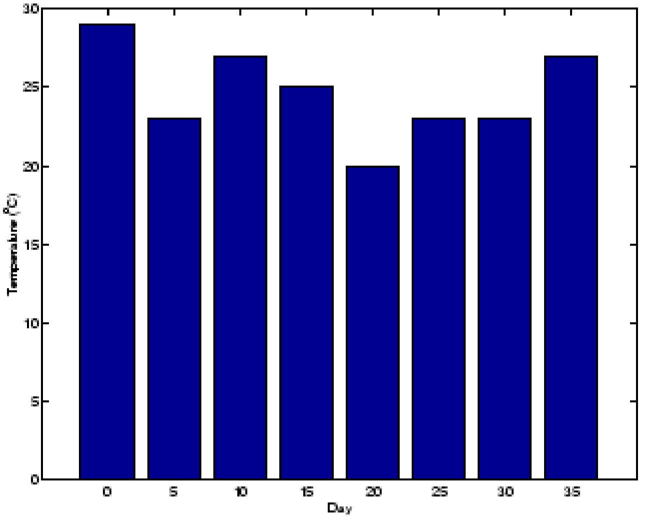
```
days = 0:5:35;
```

you can display a bar graph showing temperature measured along the *y*-axis and days along the *x*-axis using

```
bar(days,temp)
```

These statements add labels to the *x*- and *y*-axis.

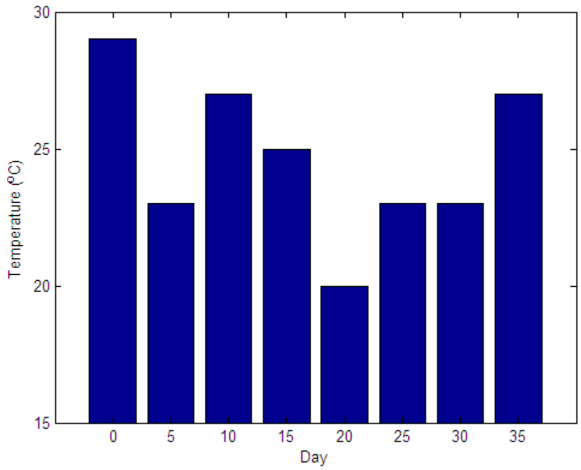
```
xlabel('Day')
ylabel('Temperature (^{o}C)')
```



Setting Y-Axis Limits

By default, the *y*-axis range is from 0 to 30. To focus on the temperature range from 15 to 30, change the *y*-axis limits.

```
set(gca,'YLim',[15 30],'Layer','top')
```



Overlaying Bar Graphs

In addition to grouping and stacking barseries, you can overlay several bars that share the same baseline and *y*-range by making each series of bars a different width and plotting the widest ones first. The following example shows how to accomplish this within an axes:

1 Define `x` and `y` data; it probably helps to make spacing of `x` values constant:

```
x=[1 3 5 7 9];
y1=[10 25 90 35 16];
K=0.5;
```

2 Plot Series 1 in blue, and set bar width to one-half an `x` unit:

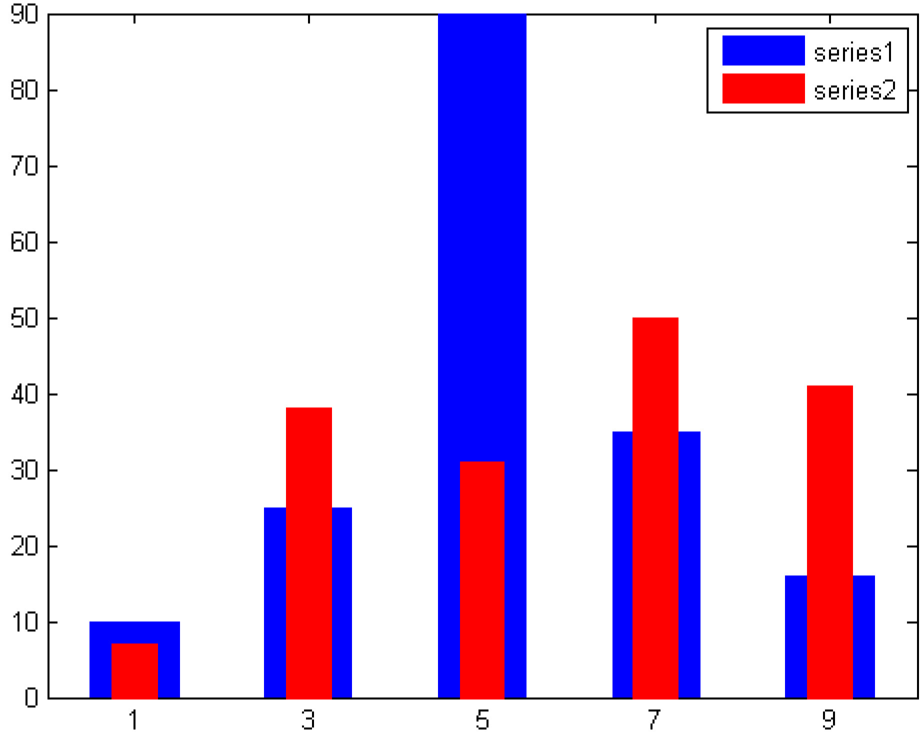
```
bar1=bar(x, y1, 'FaceColor', 'b', 'EdgeColor', 'b');
set(bar1,'BarWidth',K);
```

3 Define Series 2, and plot it in red over the first series:

```
hold on;
y2=[7 38 31 50 41];
bar2=bar(x, y2, 'FaceColor', 'r', 'EdgeColor', 'r');
```

4 Set the width of the second series to half that of the first one:

```
set(bar2,'BarWidth',K/2);
hold off;
legend('series1','series2')
```



Overlaying Other Plots on Bar Graphs

You can overlay data on a bar graph by creating another axes in the same position. This enables you to have an independent *y*-axis for the overlaid data set in contrast to the `hold on` statement, which uses the same axes.

For example, consider a bioremediation experiment that breaks down hazardous waste components into nontoxic materials. The trichloroethylene (TCE) concentration and temperature data from this experiment are

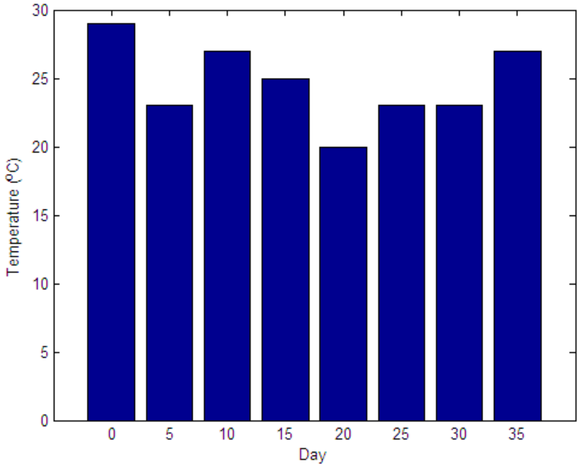
```
TCE = [515 420 370 250 135 120 60 20];
temp = [29 23 27 25 20 23 23 27];
```

This data was obtained from samples taken every five days during a thirty-five day period:

```
days = 0:5:35;
```

Display a bar graph and label the *x*- and *y*-axis using the statements

```
bar(days,temp)
xlabel('Day')
ylabel('Temperature (^{o}C)')
```



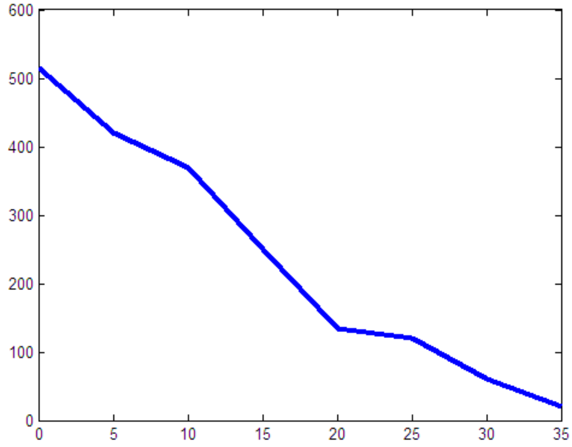
Overlaying a Line Plot on the Bar Graph

1 To overlay the concentration data on the bar graph, position a second axes at the same location as the first axes, but first save the handle of the first axes:

```
h1 = gca;
```

2 Create the second axes at the same location before plotting the second data set:

```
h2 = axes('Position',get(h1,'Position'));
plot(days,TCE,'LineWidth',3)
```

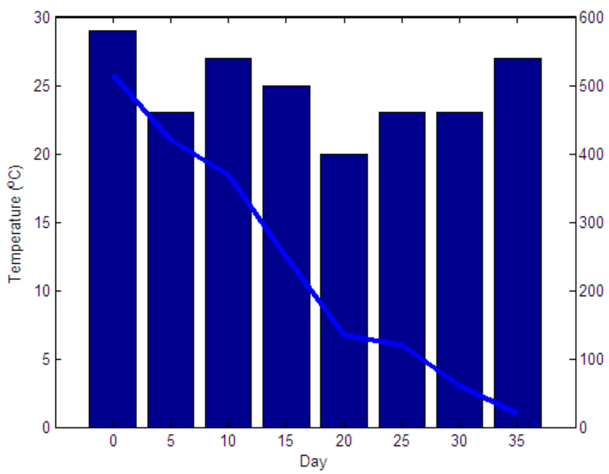


3 To ensure that the second axes does not interfere with the first, locate the *y*-axis on the right side of the axes, make the background transparent, and set the second axes' *x* tick marks to the empty matrix:

```
set(h2,'YAxisLocation','right','Color','none','XTickLabel',[])
```

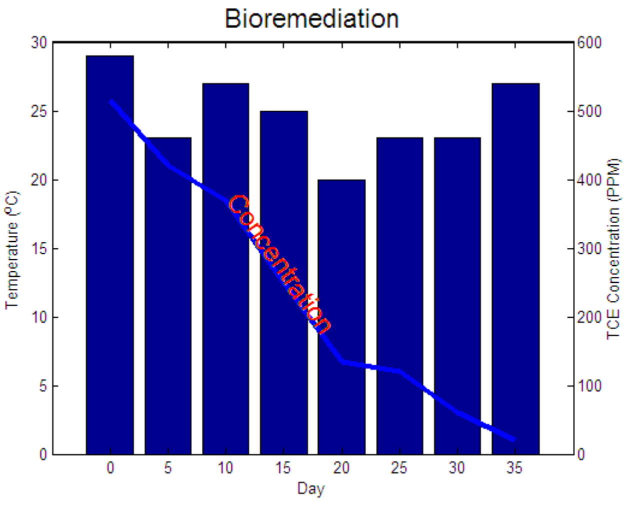
4 Align the *x*-axis of both axes and display the grid lines on top of the bars:

```
set(h2,'XLim',get(h1,'XLim'),'Layer','top')
```



Annotating the Graph. These statements annotate the graph:

```
text(11,380,'Concentration','Rotation',-55,'FontSize',16,...
     'Color','Red')
ylabel('TCE Concentration (PPM)')
title('Bioremediation','FontSize',16)
```



To print the graph, set the current figure's `PaperPositionMode` to `auto`, which ensures the printed output matches the display:

```
set(gcf,'PaperPositionMode','auto')
```

Area Graphs

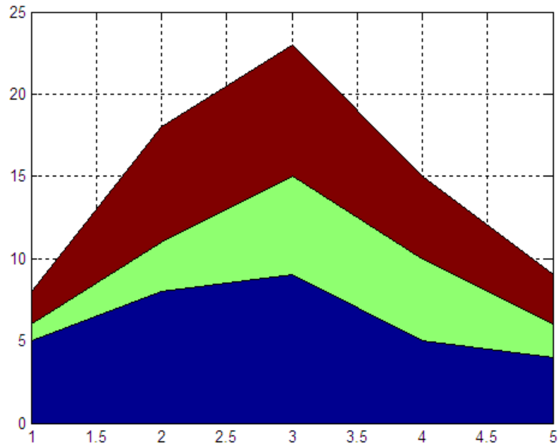
The `area` function displays curves generated from a vector or from separate columns in a matrix. `area` plots the values in each column of a matrix as a separate curve and fills the area between the curve and the *x*-axis.

Area Graphs Showing Contributing Amounts

Area graphs are useful for showing how elements in a vector or matrix contribute to the sum of all elements at a particular *x* location. By default, `area` accumulates all values from each row in a matrix and creates a curve from those values. The height of the area graph is the sum of the elements in each row. Each successive curve uses the preceding curve as its base.

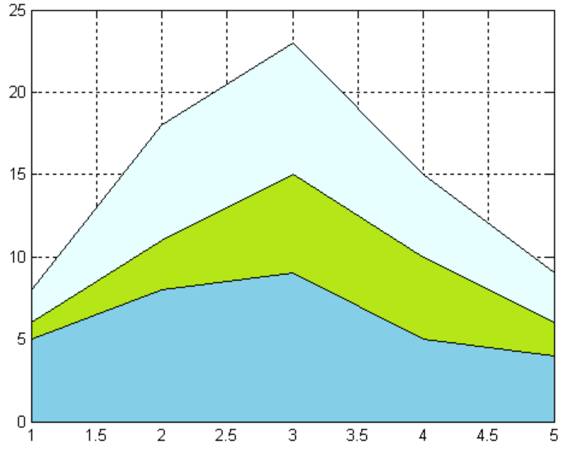
Using the matrix Y and the area function, display a graph containing three graph areas, one per column:

```
Y = [5 1 2
     8 3 7
     9 6 8
     5 5 5
     4 2 3];
harea = area(Y)
 % This returns handles to three hggroups (areaseries objects)
grid on
```



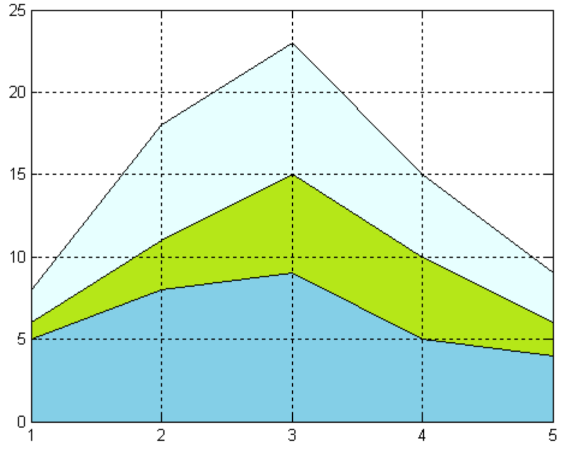
Change the face color of each layer to make the plot more readable:

```
set(get(harea(1),'Children'),'FaceColor',[.5 .8 .9])
set(get(harea(2),'Children'),'FaceColor',[.7 .9 .1])
set(get(harea(3),'Children'),'FaceColor',[.9 1 1])
```



Displaying the Grid on Top. To display the grid lines in the foreground of the area graph and display only five grid lines along the *x*-axis, use the statements

```
set(gca,'Layer','top')
set(gca,'XTick',1:5)
```



Comparing Data Sets with Area Graphs

Area graphs are useful for comparing different data sets. For example, given a vector containing sales figures

```
sales = [51.6 82.4 90.8 59.1 47.0];
```

for the five-year period

```
x = 90:94;
```

and a vector containing profits figures for the same five-year period the command

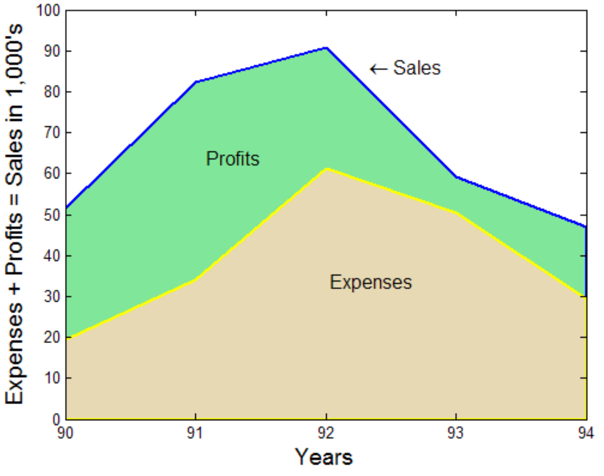
```
profits = [19.3 34.2 61.4 50.5 29.4];
```

displays both as two separate area graphs within the same axes. Set the color of the area interior (`FaceColor`), its edges (`EdgeColor`), and the width of the edge lines (`LineWidth`). See `patch` for a complete list of properties.

```
area(x,sales,'FaceColor',[.5 .9 .6],...
             'EdgeColor','b',...
             'LineWidth',2)
hold on
area(x,profits,'FaceColor',[.9 .85 .7],...
               'EdgeColor','y',...
               'LineWidth',2)
hold off
```

Annotate the graph interactively, using the `gtext` function. It accepts a string to be placed as text annotation, and enters graphic input mode. Position the cross-hair cursor where you want the lower-left corner of the text to be, and click the mouse button to complete the command. Execute the following statements to add three interactive labels and two axis labels:

```
set(gca,'XTick',[90:94])
set(gca,'Layer','top')
gtext('\leftarrow Sales')
gtext('Profits')
gtext('Expenses')
xlabel('Years','FontSize',14)
ylabel('Expenses + Profits = Sales in 1,000''s','FontSize',14)
```



100
90
80
70
60
50
40
30
20
10
0
90
91
92
93
94
Years
Expenses + Profits = Sales in 1,000's
← Sales
Profits
Expenses

Pie Charts

In this section...

Creating a Pie Chart

Pie charts are a useful way to communicate the percentage that each element in a vector or matrix contributes to the sum of all elements. `pie` and `pie3` create 2-D and 3-D pie charts. A 3-D pie chart does not show any more or different information than a 2-D pie chart does; it simply adds depth to the presentation by plotting the chart on top of a cylindrical base.

This example shows how to use the `pie` function to visualize the contribution that three products make to total sales. Given a matrix `X` where each column of `X` contains yearly sales figures for a specific product over a five-year period:

```
X = [19.3 22.1 51.6;
     34.2 70.3 82.4;
     61.4 82.9 90.8;
     50.5 54.9 59.1;
     29.4 36.3 47.0];
```

Sum each row in `X` to calculate total sales for each product over the five-year period.

```
x = sum(X);
```

You can offset the slice of the pie that makes the greatest contribution using the `explode` input argument. This argument is a vector of zero and nonzero values. Nonzero values offset the respective slice from the chart.

First, create a vector containing zeros:.

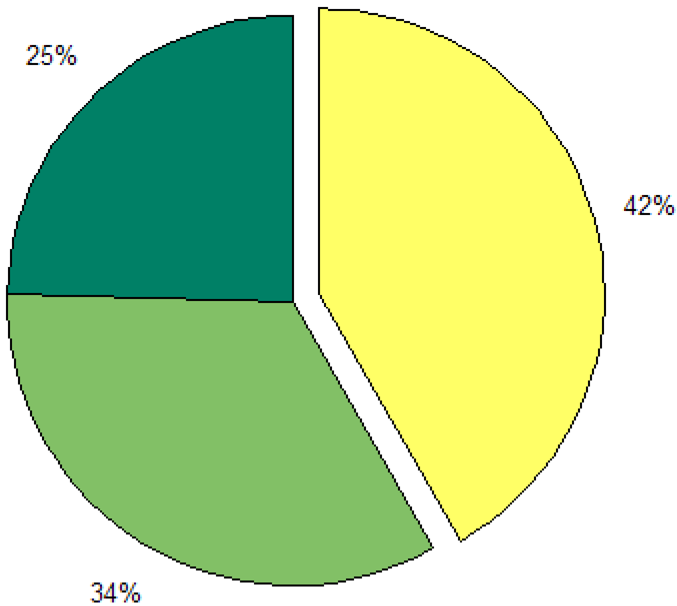
```
explode = zeros(size(x));
```

Then find the slice that contributes the most and set the corresponding `explode` element to `1`:

```
[c,offset] = max(x);
explode(offset) = 1;
```

The `explode` vector contains the elements `[0 0 1]`. To create the exploded pie chart, use the statement

```
h = pie(x,explode);
colormap summer
```



Labeling the Pie Chart

The pie chart's labels are text graphics objects. To modify the text strings and their positions, first get the objects' strings and extents. Braces around a property name ensure that `get` outputs a cell array, which is important when working with multiple objects:

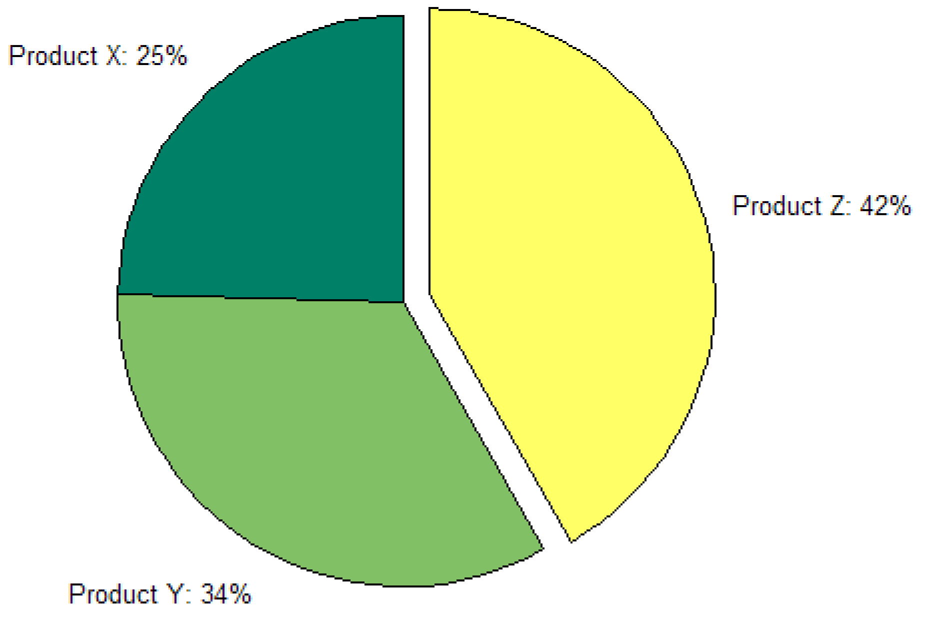
```
textObjs = findobj(h,'Type','text');
oldStr = get(textObjs,{'String'});
val = get(textObjs,{'Extent'});
oldExt = cat(1,val{:});
```

Create the new strings, and set the text objects' `String` properties to the new strings:

```
Names = {'Product X: ';'Product Y: ';'Product Z: '};
newStr = strcat(Names,oldStr);
set(textObjs,{'String'},newStr)
```

Find the difference between the widths of the new and old text strings and change the values of the `Position` properties:

```
val1 = get(textObjs, {'Extent'});
newExt = cat(1, val1{:});
offset = sign(oldExt(:,1)).*(newExt(:,3)-oldExt(:,3))/2;
pos = get(textObjs, {'Position'});
textPos =  cat(1, pos{:});
textPos(:,1) = textPos(:,1)+offset;
set(textObjs,{'Position'},num2cell(textPos,[3,2]))
```

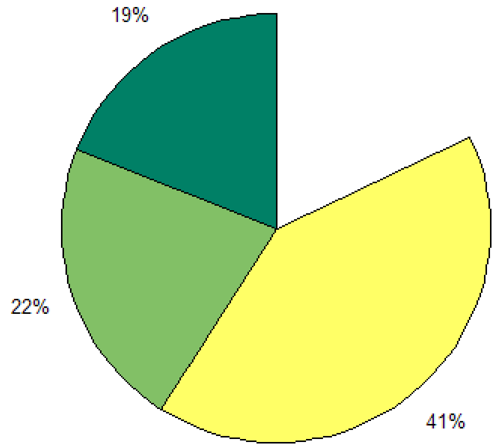


Removing a Piece from a Pie Chart

When the sum of the elements in the first input argument is equal to or greater than 1, pie and pie3 normalize the values. So, given a vector of elements x, each slice has an area of $x_i/sum(x_i)$, where x_i is an element of x. The normalized value specifies the fractional part of each pie slice.

When the sum of the elements in the first input argument is less than 1, pie and pie3 do not normalize the elements of vector x. They draw a partial pie.

```
x = [.19 .22 .41];
pie(x)
colormap summer
```


19%
22%
41%

Histograms

In this section...

Functions for Creating Histograms

Histograms show the distribution of data values across a data range. They do this by dividing the data range into a certain number of intervals (called "binning" the data), tabulating the number of values that fall into each interval (or "bin"), and plotting the values in the bins using bars or wedges of varying height. The functions that create histograms are `hist` and `rose`.

Function	Description
`hist`	Displays data in a Cartesian coordinate system.
`rose`	Displays data in a polar coordinate system.

You can specify the number of bins to use as a scalar second argument. If omitted, the default is 10 (`hist`) or 20 (`rose`). Data values passed to `hist` can be in any units and can be *n*-by-*m*, but `rose` expects values to be in radians in a 1-by-*n* or *n*-by-1 vector. The height (or length when using `rose`) of the bins represents the number of values that fall in each bin. You can also vary the size of bins by specifying a vector for apportioning bin widths as the second argument.

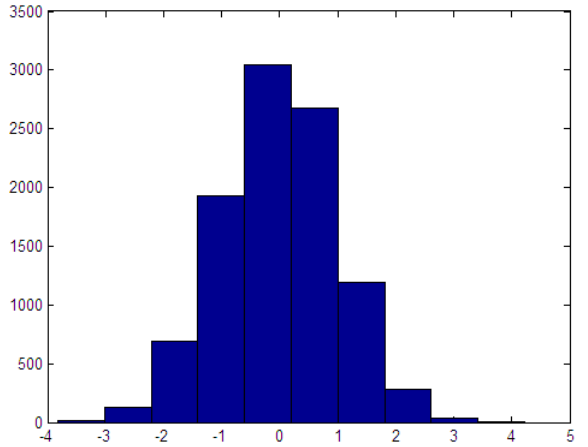
Histograms in Cartesian Coordinates

The `hist` function shows the distribution of the elements in `Y` as a histogram with equally spaced bins between the minimum and maximum values in `Y`. If `Y` is a vector and is the only argument, `hist` creates up to 10 bins. For example:

```
yn = randn(10000,1);
```

```
hist(yn)
```

generates 10,000 random numbers and creates a histogram with 10 bins distributed along the *x*-axis between the minimum and maximum values of `yn`.

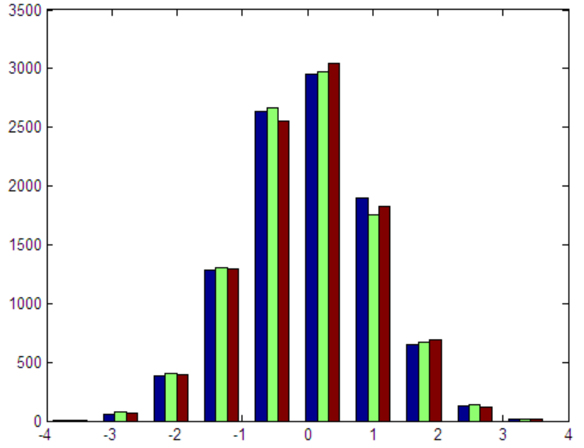


Matrix Input Argument

When `Y` is a matrix, `hist` creates a set of bins for each column, displaying each set in a separate color. The statements

```
Y = randn(10000,3);
hist(Y)
```

create a histogram showing 10 bins for each column in `Y`.



Histograms in Polar Coordinates

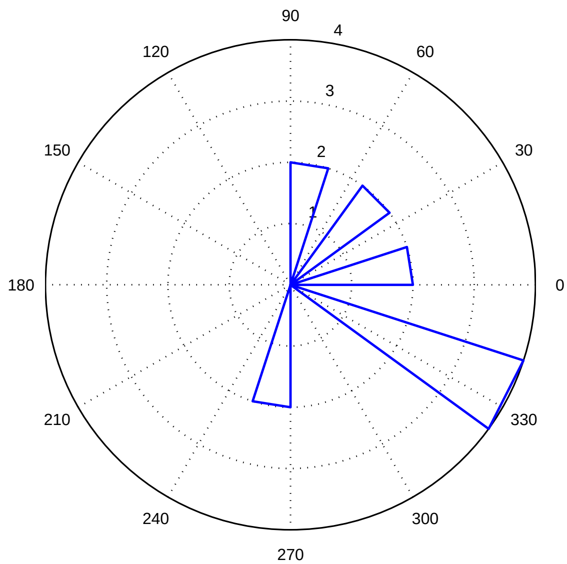
A `rose` plot is a histogram created in a polar coordinate system. For example, consider samples of the wind direction taken over a 12-hour period:

```
wdir = [45 90 90 45 360 335 360 270 335 270 335 335];
```

To display this data using the `rose` function, convert the data to radians, and then use the data as an argument to the `rose` function. Increase the `LineWidth` property of the line to improve the visibility of the plot (`findobj`):

```
wdir = wdir * pi/180;
rose(wdir)
hline = findobj(gca,'Type','line');
set(hline,'LineWidth',1.5)
```

The plot shows that the wind direction was primarily 335° during the 12-hour period.



Specifying Number of Bins

`hist` and `rose` interpret their second argument in one of two ways—as the locations on the axis or the number of bins. When the second argument is a vector `x`, it specifies the locations on the axis and distributes the elements in `length(x)` bins. When the second argument is a scalar `x`, `hist` and `rose` distribute the elements in `x` bins.

For example, compare the distribution of data created by two MATLAB functions that generate random numbers. The `randn` function generates normally distributed random numbers, whereas the `rand` function generates uniformly distributed random numbers:

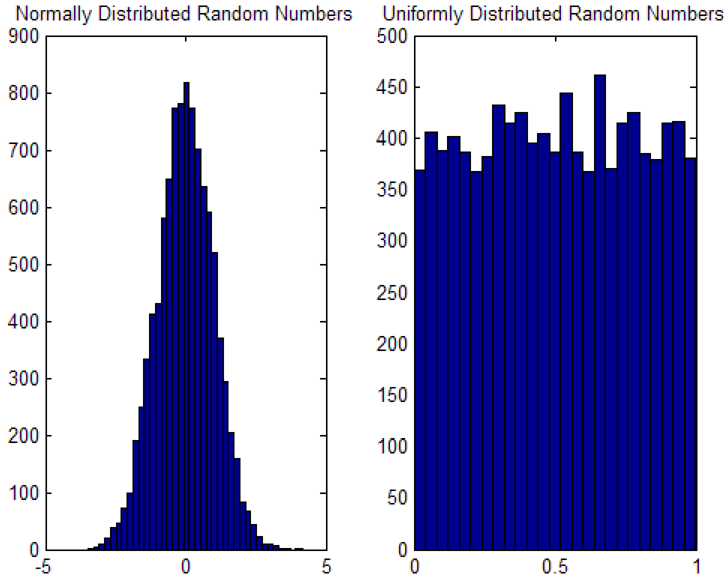
```
yn = randn(10000,1);
yu = rand(10000,1);
```

The first histogram displays the data distribution resulting from the `randn` function. The locations on the *x*-axis and number of bins depend on the vector `x`.

```
x = min(yn):.2:max(yn);
subplot(1,2,1)
hist(yn,x)
title('Normally Distributed Random Numbers')
```

The second histogram displays the data distribution resulting from the `rand` function and explicitly creates 25 bins along the *x*-axis.

```
subplot(1,2,2)
hist(yu,25)
title('Uniformly Distributed Random Numbers')
```



Note You can change the aspect ratio of the histogram plots using the mouse to resize the figure window. However, before creating hardcopy output, set the figure's `PaperPositionMode` to `auto` to produce printed output that matches the display.

```
set(gcf,'PaperPositionMode','auto')
```

Using Data Cursors with Histograms

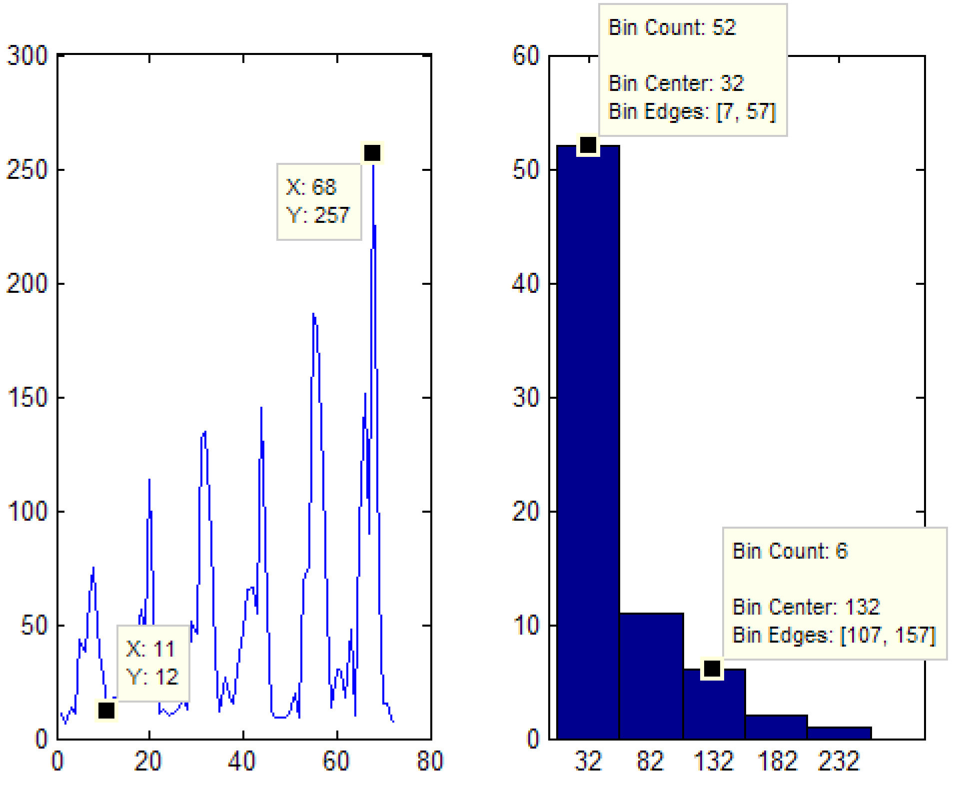
When you use the Data Cursor tool  on a histogram plot, it customizes the data tips it displays in an appropriate way. Instead of providing *x*-, *y*-,*z*-coordinates, the datatips display the following information:

- Number of observations falling into the selected bin
- The *x* value of the bin's center
- The lower and upper *x* values for the bin

For example, The following figures show a line plot and a histogram of `count.dat`, a demo data set that contains three columns, giving hourly traffic counts at three different locations. The plots depict the sum the values over the locations. Each graph displays two datatips, but the datatips in the right-hand plot give information specific to histograms.

```
load count.dat
figure;
subplot(1,2,1); plot(count(:))
subplot(1,2,2); hist(count(:),5)
datacursormode on
```

Click to place a datatip or drag an existing one to a new location. You can add new datatips to a plot by right-clicking, selecting **Create new datatip**, and clicking the graph where you want to put it.



When you add datatips to histograms or bar graphs showing groups of data, you can move a datatip to any other bar by clicking inside that bar. If you use the cursor keys to shift a datatip back or forth across the graph, the datatip moves to the preceding or succeeding bar of the same color.

Discrete Data Graphs

In this section...

Functions for Creating Graphs of Discrete Data

In addition to bar graphs and pie charts, specialized MATLAB graphics functions appropriately display discrete data. Discrete data generally represents counts of things, such as traffic accidents by month or components produced or rejected during the course of a production run. This section describes how to use stem plots and stairstep plots to display this type of data. The functions for generating discrete data graphs provided are

Function	Description
`stem`	Displays a discrete sequence of *y*-data as stems from *x*-axis.
`stem3`	Displays a discrete sequence of *z*-data as stems from *xy*-plane.
`stairs`	Displays a discrete sequence of *y*-data as steps from *x*-axis.

Two-Dimensional Stem Plots

A stem plot displays data as lines (stems) terminated with a marker symbol at each data value. In a 2-D graph, stems extend from the *x*-axis.

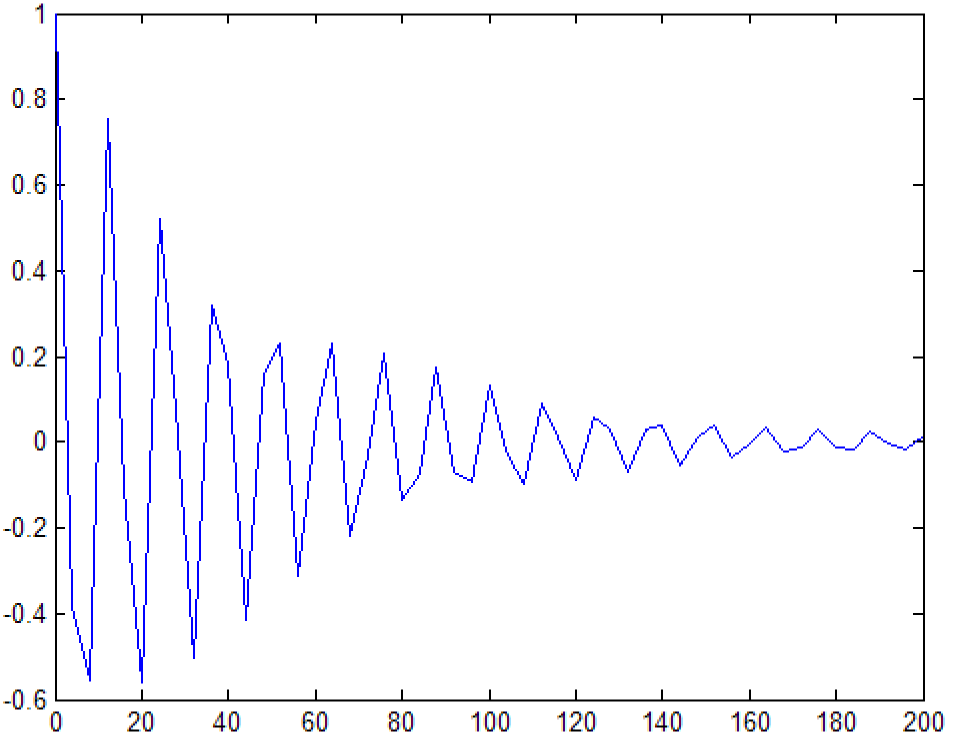
The `stem` function displays two-dimensional discrete sequence data. For example, evaluating the function $y = e^{-\alpha t} \cos \beta t$ with the values

```
alpha = .02; beta = .5; t = 0:4:200;
```

```
y = exp(-alpha*t).*cos(beta*t);
```

yields a vector of discrete values for `y` at given values of `t`. A line plot shows the data points connected with a straight line.

```
plot(t,y)
```

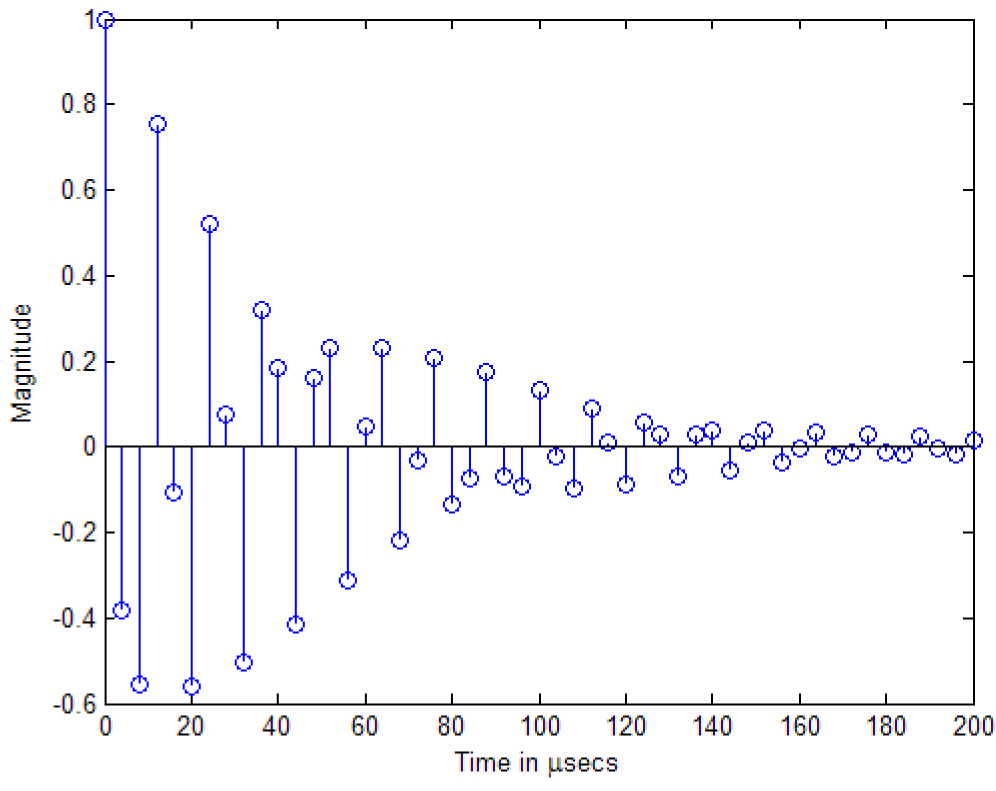


A stem plot of the same function plots only discrete points on the curve:

```
stem(t,y)
```

Add axes labels to the *x*- and *y*-axis:

```
xlabel('Time in \musecs')
ylabel('Magnitude')
```

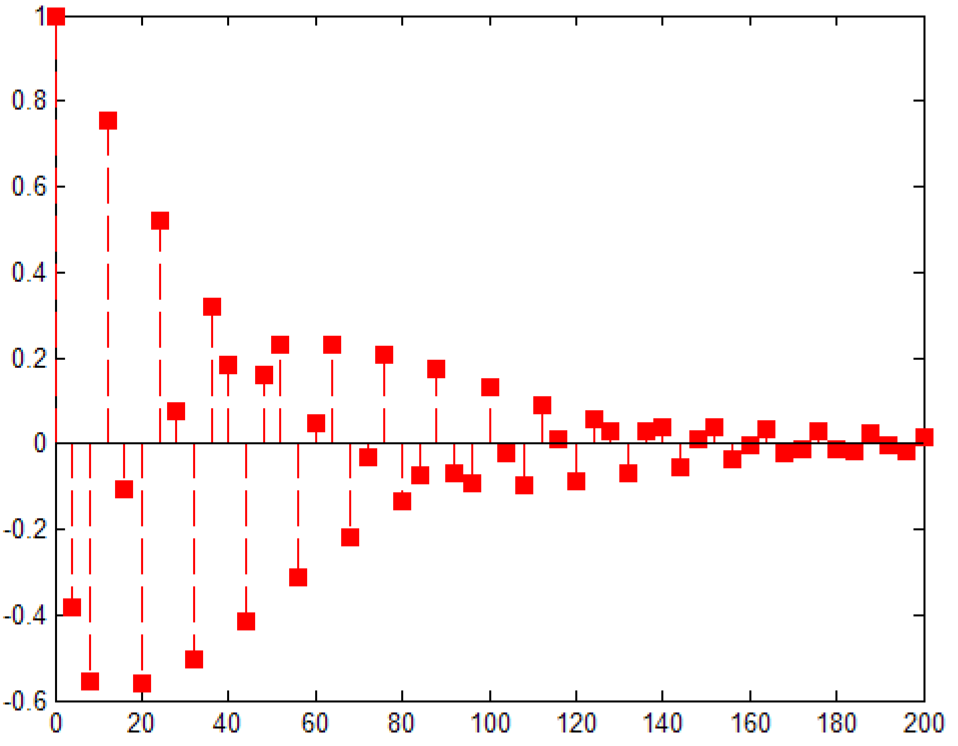


If you specify only one argument, the number of samples is equal to the length of that argument. In this example, the number of samples is a function of `t`, which contains 51 elements and determines the length of `y`.

Customizing the Graph

You can specify the line style, the type of marker, and the color used in the stem plot. For example, adding the string '--sr' specifies a dashed line (--), a square marker (s), and a red color (r). The 'fill' argument colors the face of the marker.

```
stem(t,y,'--sr','fill')
```



Setting the aspect ratio of the *x*- and *y*-axis to 2:1 improves the utility of the graph. You can do this by setting the aspect ratio of the plot box using `pbaspect`:

```
pbaspect([2,1,1])
```

This is equivalent to setting the `PlotBoxApectRatio` property directly:

```
set(gca,'PlotBoxAspectRatio',[2,1,1])
```

See `LineSpec` for a list of line styles and marker types.

Combining Stem Plots with Line Plots

Sometimes it is useful to display more than one plot simultaneously with a stem plot to show how you arrived at a result. For example, create a linearly spaced vector with 60 elements and define two functions, `a` and `b`:

```
x = linspace(0,2*pi,60);
a = sin(x);
b = cos(x);
```

Create a stem plot showing the linear combination of the two functions:

```
stem_handles = stem(x,a+b);
```

Overlaying `a` and `b` as line plots helps visualize the functions. Before plotting the two curves, set `hold` to `on` so the stem plot remains displayed:

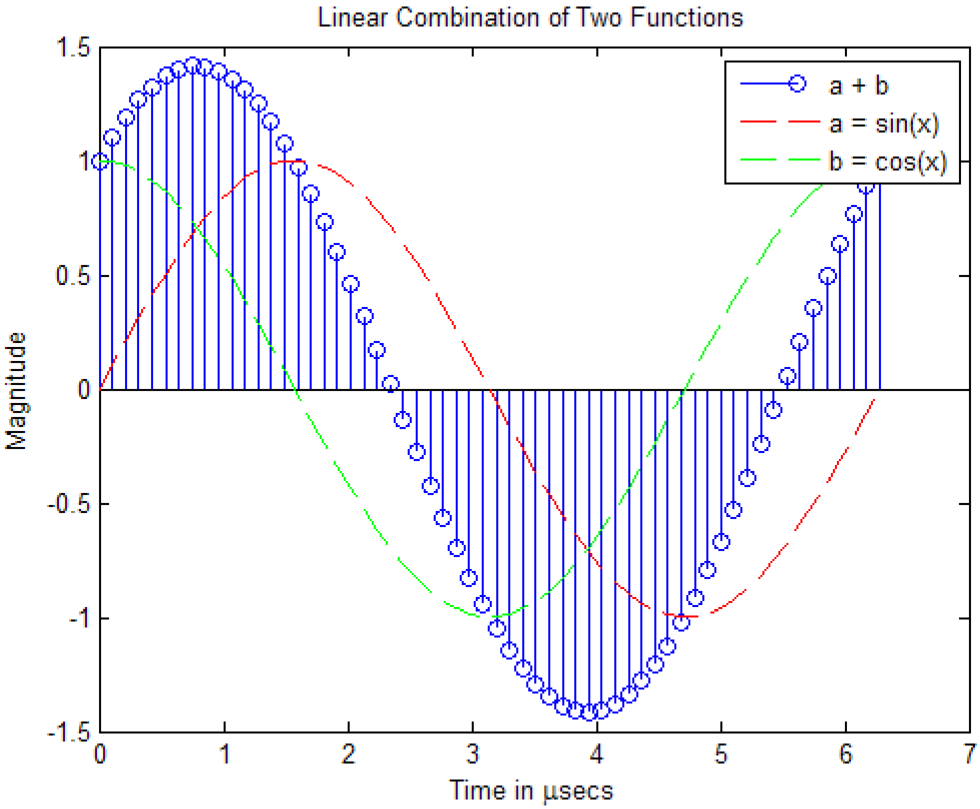
```
hold on
plot_handles = plot(x,a,'--r',x,b,'--g');
hold off
```

Use `legend` to annotate the graph. The stem and plot handles passed to `legend` identify the lines to label. Stem plots are composed of two lines; one draws the markers and the other draws the vertical stems. To create the legend, use the first handle returned by `stem`, which identifies the marker line:

```
legend_handles = [stem_handles(1);plot_handles];
legend(legend_handles,'a + b','a = sin(x)','b = cos(x)')
```

Labeling the axes and creating a title finishes the graph:

```
xlabel('Time in \musecs')
ylabel('Magnitude')
title('Linear Combination of Two Functions')
```



Three-Dimensional Stem Plots

`stem3` displays 3-D stem plots extending from the *xy*-plane. With only one vector argument, the stems are plotted in one row at `x = 1` or `y = 1`, depending on whether the argument is a column or row vector. `stem3` is intended to display data that you cannot visualize in a 2-D view.

Example — 3-D Stem Plot of an FFT

Fast Fourier transforms are calculated at points around the unit circle on the complex plane. It is interesting to visualize the plot around the unit circle. Calculating the unit circle

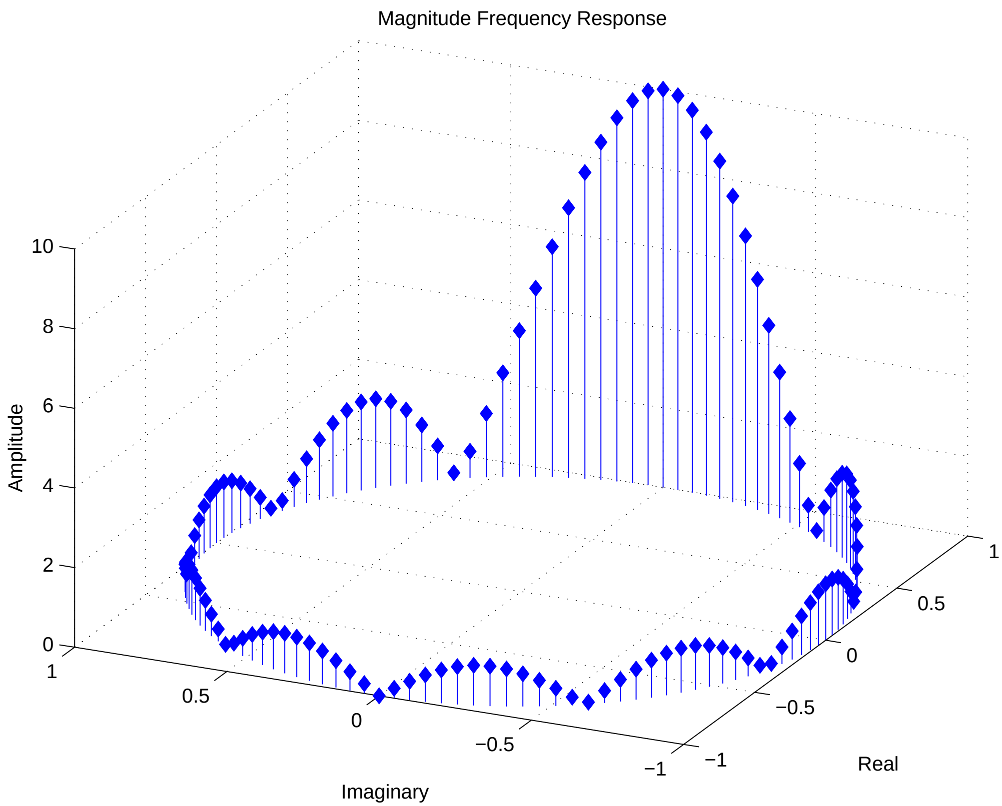
```
th = (0:127)/128*2*pi;
x = cos(th);
y = sin(th);
```

and the magnitude frequency response of a step function. The command

```
f = abs(fft(ones(10,1),128));
```

displays the data using a 3-D stem plot, terminating the stems with filled diamond markers:

```
stem3(x,y,f','d','fill')
view([-65 30])
```



Label the Graph

Label the graph with the statements

```
xlabel('Real')
ylabel('Imaginary')
zlabel('Amplitude')
title('Magnitude Frequency Response')
```

To change the orientation of the view, turn on mouse-based 3-D rotation:

```
rotate3d on
```

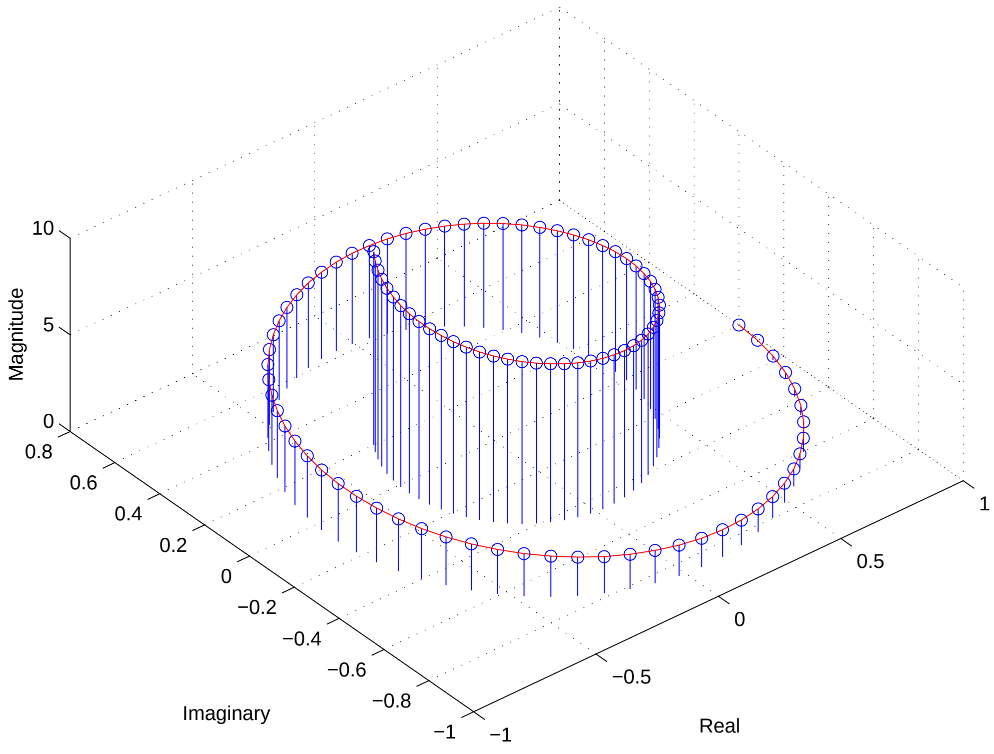
Example — Combining Stem and Line Plots

Three-dimensional stem plots work well for visualizing discrete functions that do not output a large number of data points. For example, use stem3 to visualize the Laplace transform basis function, $y = e^{-st}$, for a particular constant value of s:

```
t = 0:.1:10; % Time limits
s = 0.1+i; % Spiral rate
y = exp(-s*t); % Compute decaying exponential
```

Using t as magnitudes that increase with time, create a spiral with increasing height and draw a curve through the tops of the stems to improve definition:

```
stem3(real(y),imag(y),t)
hold on
plot3(real(y),imag(y),t,'r')
hold off
view(-39.5,62)
```



Label the Graph

Add axes labels with the statements

```
xlabel('Real')
ylabel('Imaginary')
zlabel('Magnitude')
```

Stairstep Plots

Stairstep plots display data as the leading edges of a constant interval (i.e., zero-order hold state). This type of plot holds the data at a constant *y* value for all values between *x(i)* and *x(i+1)*, where *i* is the index into the *x* data.

This type of plot is useful for drawing time-history plots of digitally sampled data systems.

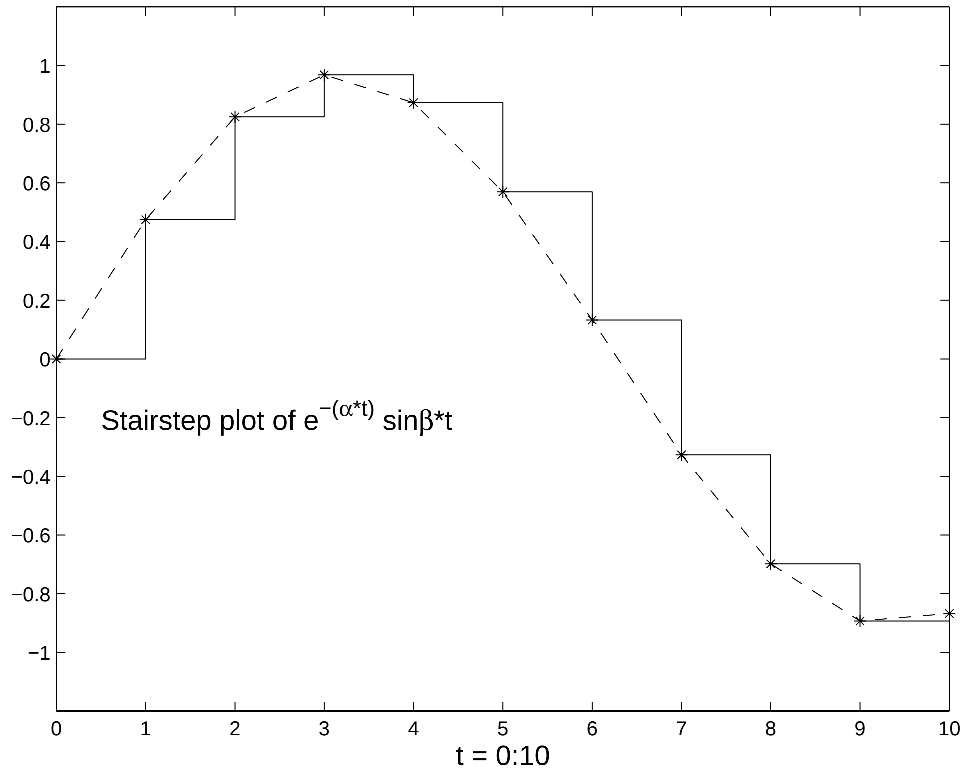
Example — Stairstep Plot of a Function

Define a function f that varies over time:

```
alpha = 0.01;
beta = 0.5;
t = 0:10;
f = exp(-alpha*t).*sin(beta*t);
```

Use stairs to display the function as a stairstep plot and a linearly interpolated function:

```
stairs(t,f)
hold on
plot(t,f,'--*')
hold off
```



Annotate the graph and set the axes limits:

```
label = 'Stairstep plot of e^{-(\alpha*t)} sin\beta*t';
text(0.5,-0.2,label,'FontSize',14)
xlabel('t = 0:10','FontSize',14)
axis([0 10 -1.2 1.2])
```

Direction and Velocity Vector Graphs

In this section...

Functions for Graphing Vector Quantities

Four MATLAB functions display data consisting of direction vectors and velocity vectors; three create 2-D plots and one creates 3-D plots.

Function	Description
`compass`	Displays vectors emanating from the origin of a polar plot.
`feather`	Displays vectors extending from equally spaced points along a horizontal line.
`quiver`	Displays 2-D vectors specified by (u,v) components.
`quiver3`	Displays 3-D vectors specified by (u,v,w) components.

For feather and compass plots, you define the vectors using one or two arguments. The arguments specify the u and v components of the vectors relative to the origin. If you specify two arguments, the first specifies the u components of the vectors, and the second specifies the v components of the vectors. If you specify one argument, the functions treat the elements as complex numbers. The real parts are the u components, and the imaginary parts are the v components.

For quiver plots, in addition to the u-v components, you also specify x,y locations (or x,y,z locations in the case of `quiver3`) to establish an origin for each vector.

Compass Plots

The `compass` function shows vectors emanating from the origin of a graph. The function takes Cartesian coordinates and plots them on a circular grid.

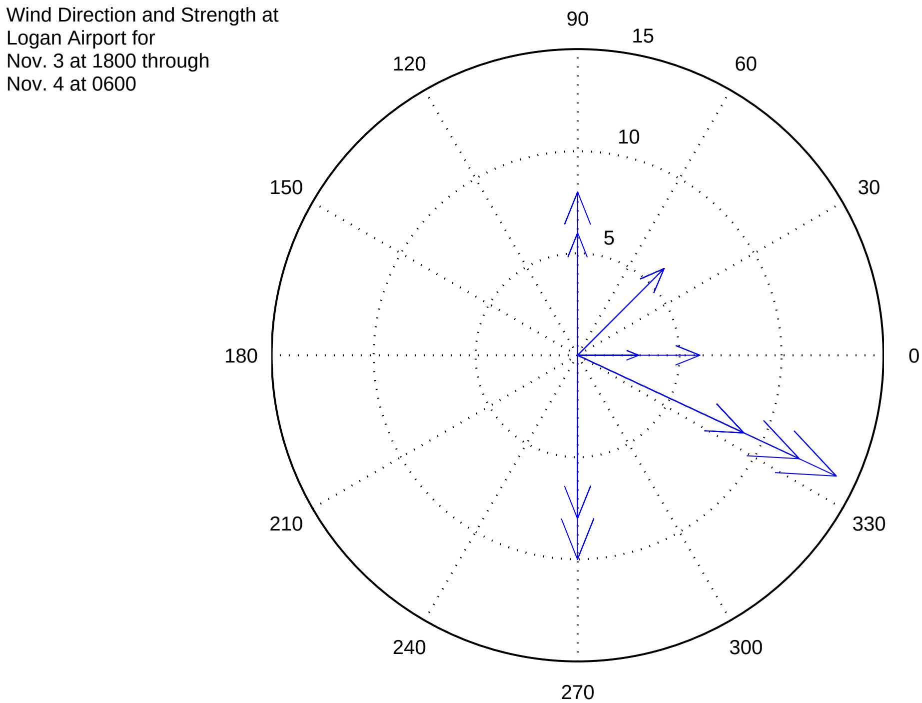
Example — Compass Plot of Wind Direction and Speed

This example shows a compass plot indicating the wind direction and strength during a 12-hour period. Two vectors define the wind direction and strength:

```
wdir = [45 90 90 45 360 335 360 270 335 270 335 335];
knots = [6 6 8 6 3 9 6 8 9 10 14 12];
```

Convert the wind direction, given as angles, into radians before converting the wind direction into Cartesian coordinates:

```
rdir = wdir * pi/180;
[x,y] = pol2cart(rdir,knots);
compass(x,y)
```



Create text to annotate the graph:

```
desc = {'Wind Direction and Strength at',
        'Logan Airport for ',
        'Nov. 3 at 1800 through',
        'Nov. 4 at 0600'};
text(-28,15,desc)
```

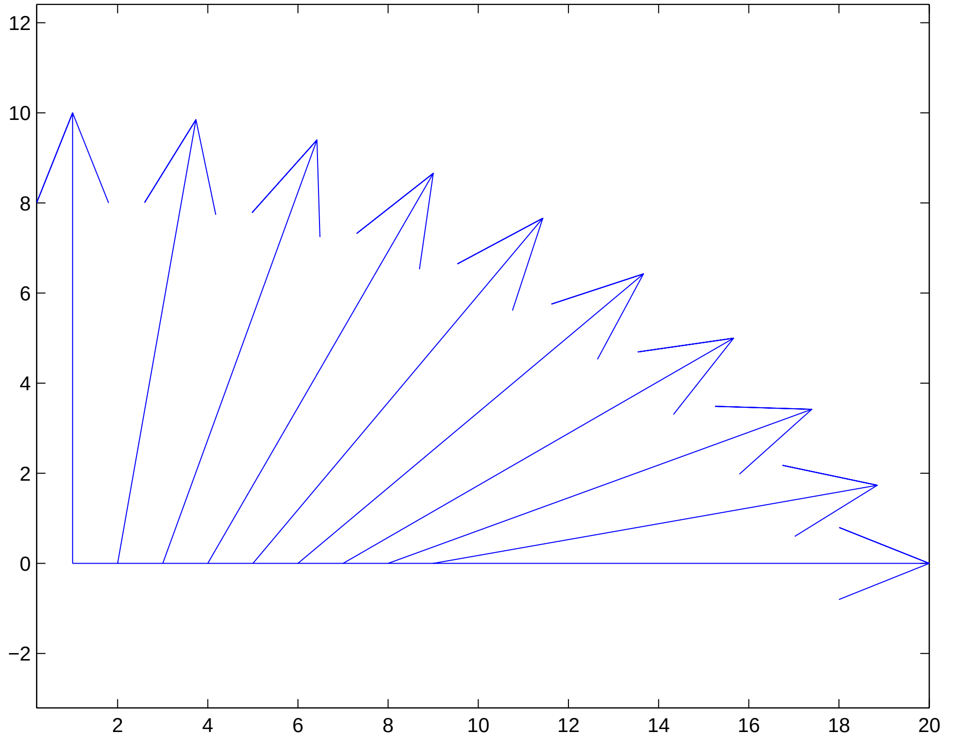
Feather Plots

The `feather` function shows vectors emanating from a straight line parallel to the x-axis. For example, create a vector of angles from 90° to 0° and a vector the same size, with each element equal to `1`.

```
theta = 90:-10:0;
r = ones(size(theta));
```

Before creating a feather plot, transform the data into Cartesian coordinates and increase the magnitude of r to make the arrows more distinctive:

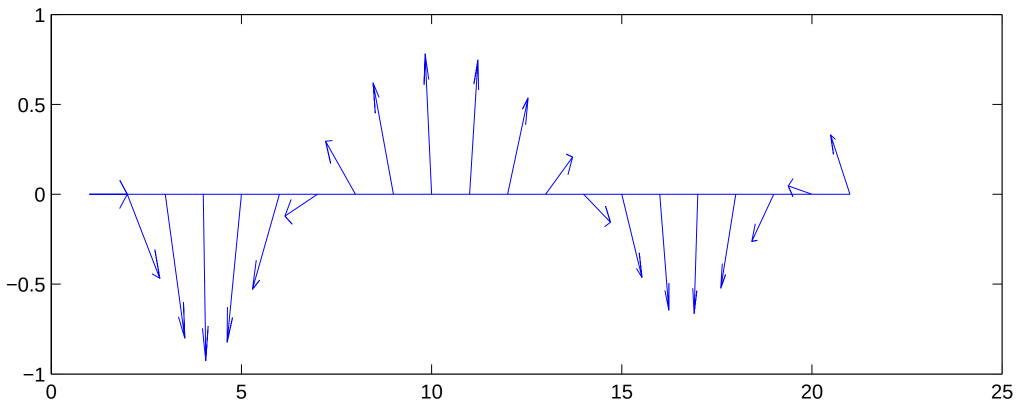
```
[u,v] = pol2cart(theta*pi/180,r*10);
feather(u,v)
axis equal
```



Plotting Complex Numbers

If the input argument Z is a matrix of complex numbers, feather interprets the real parts of Z as the *x* components of the vectors and the imaginary parts as the *y* components of the vectors:

```
t = 0:0.5:10;     % Time limits
s = 0.05+i;       % Spiral rate
Z = exp(-s*t);    % Compute decaying exponential
feather(Z)
```



Printing the Graph

This particular graph looks better if you change the figure's aspect ratio by stretching the figure lengthwise using the mouse. However, to maintain this shape in the printed output, set the figure's PaperPositionMode to auto.

```
set(gcf,'PaperPositionMode','auto')
```

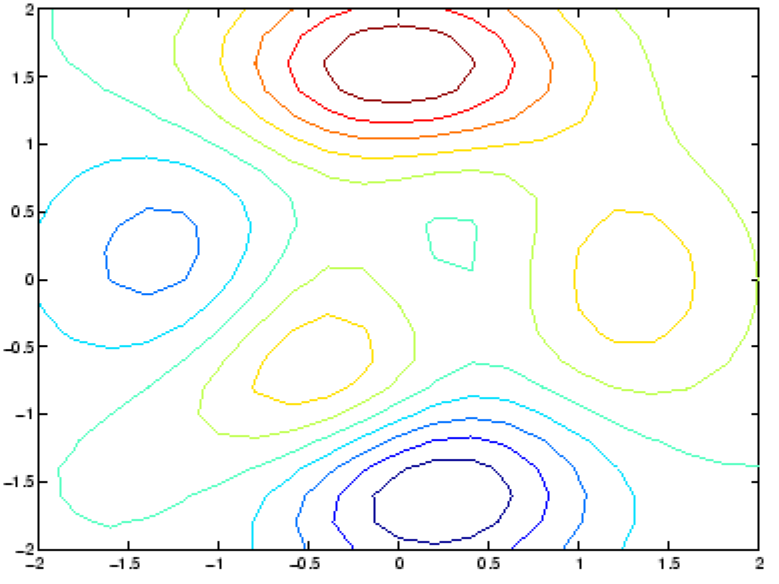
In this mode, MATLAB prints the figure as it appears on screen.

Two-Dimensional Quiver Plots

The quiver function shows vectors at given points in two-dimensional space. The *x* and *y* components define the vectors.

A quiver plot is useful when displayed with another plot. For example, create 10 contours of the peaks function. (See "Contour Plots" on page 5-57 for more information.)

```
n = -2.0:.2:2.0;
[X,Y,Z] = peaks(n);
contour(X,Y,Z,10)
```

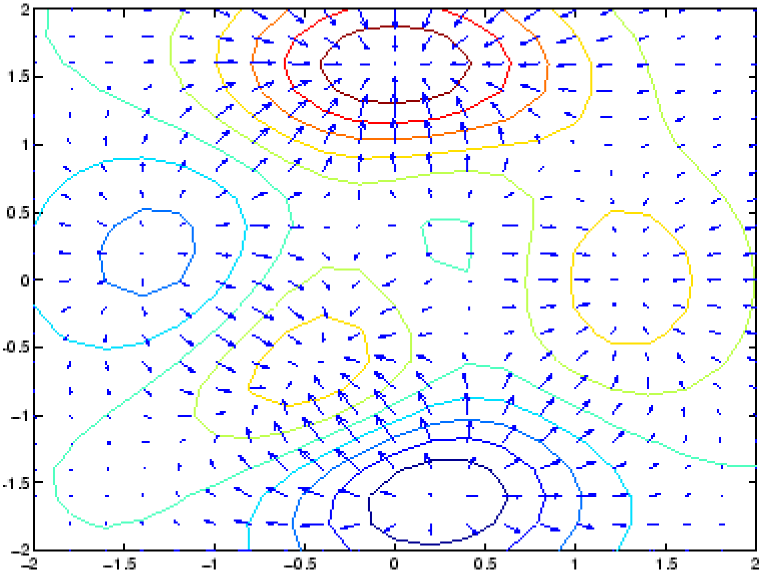


Now use gradient to create the vector components to use as inputs to quiver:

```
[U,V] = gradient(Z,.2);
```

Set hold to on and add the contour plot:

```
hold on
quiver(X,Y,U,V)
hold off
```



Three-Dimensional Quiver Plots

Three-dimensional quiver plots (quiver3) display vectors consisting of (u,v,w) components at (x,y,z) locations. For example, you can show the path of a projectile as a function of time, $z(t) = v_z t + \frac{at^2}{2}$

Assign values to the constants vz and a:

```
vz = 10;             % Velocity
a = -32;             % Acceleration
```

Calculate the height z as time varies from 0 to 1 in increments of 0.1:

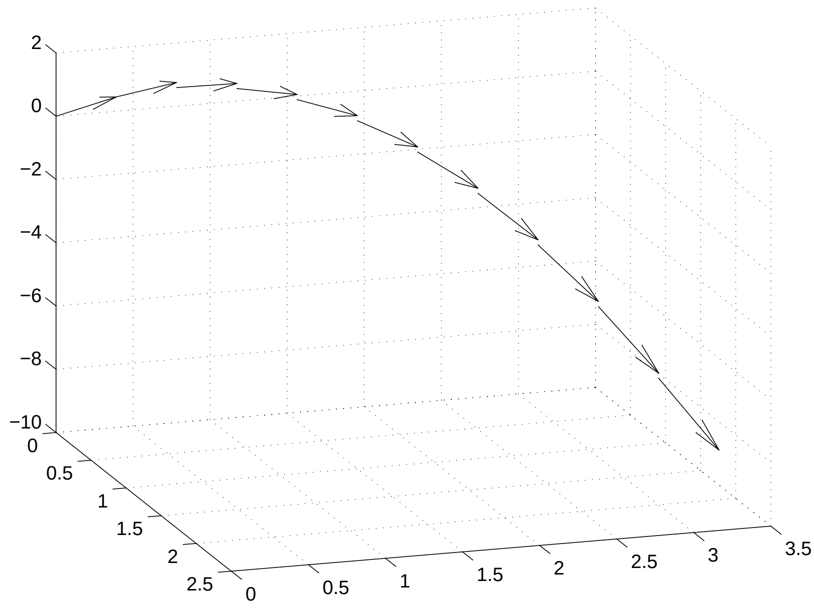
```
t = 0:.1:1;
z = vz*t + 1/2*a*t.^2;
```

Calculate the position in the *x* and *y* directions:

```
vx = 2;
x = vx*t;
vy = 3;
y = vy*t;
```

Compute the components of the velocity vectors and display the vectors using the 3-D quiver plot:

```
u = gradient(x);
v = gradient(y);
w = gradient(z);
scale = 0;
quiver3(x,y,z,u,v,w,scale)
view([70 18])
```



Contour Plots

In this section...

Functions for Creating Contour Displays

The contouring display functions compute, plot, and label isolines (contour lines) for one or more matrices. These displays vary according to whether they plot plain contour lines, filled contour lines, raised contours, or contours in concert with mesh or surface plots. Two of the functions support contouring. The low-level `contourc` function computes isolines but does not plot them. The `clabel` function places elevation labels on previously generated contours.

Function	Description
`contour`	Displays 2-D isolines generated from values given by a matrix Z.
`contour3`	Displays 3-D isolines generated from values given by a matrix Z.
`contourf`	Displays a 2-D contour plot and fills the area between the isolines with a solid color.
`contourc`	Low-level function to calculate the contour matrix used by the other contour functions.

Function	Description
meshc	Creates a mesh plot with a corresponding 2-D contour plot.
surfc	Creates a surface plot with a corresponding 2-D contour plot.
clabel	Generates labels using the contour matrix returned from calling the contouring function and displays the labels in the current figure.

Creating Simple Contour Plots

`contour` and `contour3` display 2-D and 3-D contours, respectively. They can be called with separate *x*, *y*, and *z* matrices, but need only one input argument—a *z* matrix interpreted as heights with respect to a plane. In this case, the contour functions determine the number of contours to display based on the minimum and maximum data values.

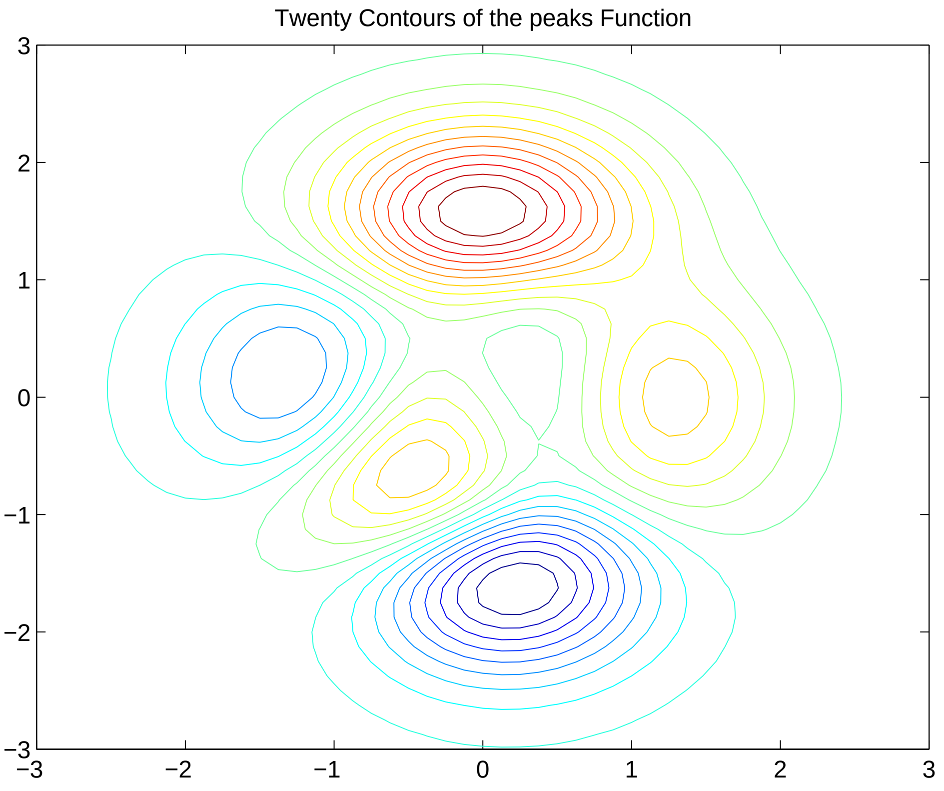
To explicitly set the number of contour levels displayed by the functions, you specify a second optional argument.

Contour Plot of the Peaks Function

The statements

```
[X,Y,Z] = peaks;
contour(X,Y,Z,20)
```

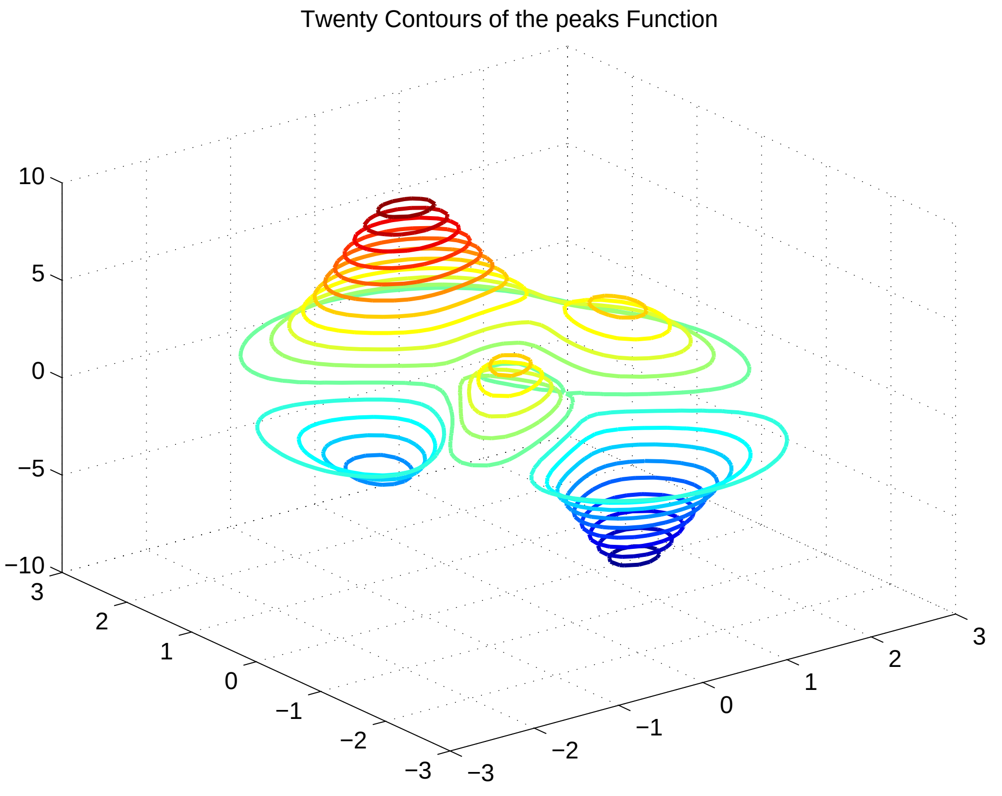
display 20 contours of the `peaks` function in a 2-D view.



The statements

```
[X,Y,Z] = peaks;
contour3(X,Y,Z,20)
h = findobj('Type','patch');
set(h,'LineWidth',2)
title('Twenty Contours of the peaks Function')
```

display 20 contours of the `peaks` function in a 3-D view and increase the line width to 2 points.



Labeling Contours

Each contour level has a value associated with it. `clabel` uses these values to display labels for 2-D contour lines. The contour matrix contains the values `clabel` uses for the labels. This matrix is returned by `contour`, `contour3`, and `contourf` and is described in "The Contouring Algorithm" on page 5-70.

`clabel` optionally returns the handles of the text objects used as labels. You can then use these handles to set the properties of the label string.

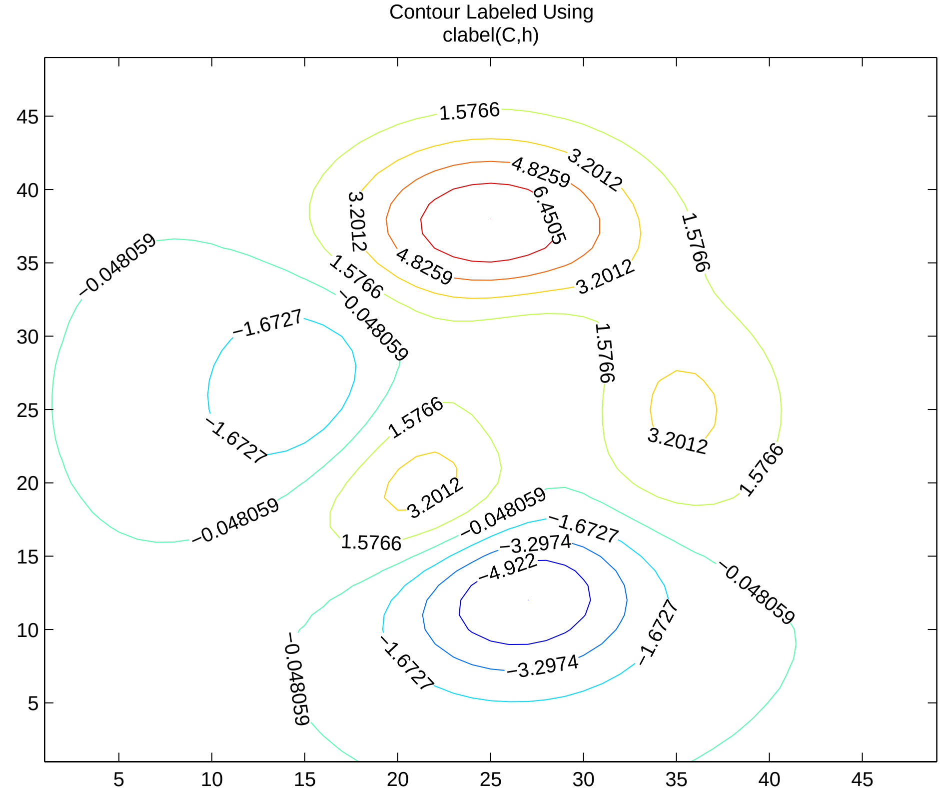
For example, display 10 contour levels of the `peaks` function:

```
Z = peaks;
[C,h] = contour(Z,10);
```

Label the contours and display a title:

```
clabel(C,h)
title({'Contour Labeled Using','clabel(C,h)'})
```

`clabel` labels only those contour lines that are large enough to have an inline label inserted.



The `'manual'` option enables you to add labels by selecting the contour you want to label with the mouse.

You can also use this option to label only those contours you select interactively.

For example:

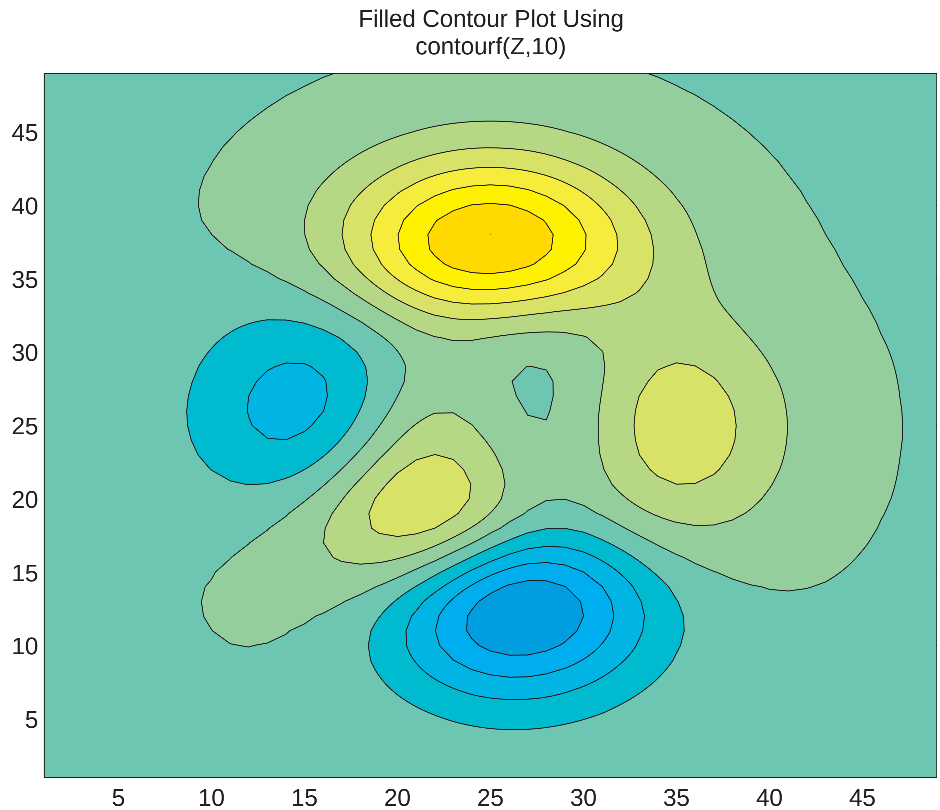
```
clabel(C,h,'manual')
```

displays a crosshair cursor when your cursor is inside the figure. Pressing any mouse button labels the contour line closest to the center of the crosshair.

Filled Contours

The `contourf` displays a two-dimensional contour plot and fills the areas between contour lines. Use `caxis` to control the mapping of contour to color. For example, this filled contour plot of the peaks data uses `caxis` to map the fill colors into the center of the colormap:

```
Z = peaks;
[C,h] = contourf(Z,10);
caxis([-20 20])
title({'Filled Contour Plot Using','contourf(Z,10)'})
```



Specifying Contour Levels

The contouring functions permit you to specify the number of contour levels or the particular contour levels to draw. In the case of `contour`, the two forms of the function are `contour(Z,n)` and `contour(Z,v)`. `Z` is the data matrix, `n` is the number of contour lines, and `v` is a vector of specific contour levels.

When you specify `n` (the number of contour levels to plot), you are setting the `LevelStep` property of the contourgroup. If you want to always draw `n` contour levels even when the data range of `Z` changes, obtain the contourgroup's handle and set its `LevelStepMode` to `'manual'`.

When you specify `v` (a vector itemizing contour levels), you are setting the contourgroup's `LevelList` property. By default, the `LevelList` is recomputed whenever contours are redrawn. If you always want to depict the same contour levels, even if the data changes, set the contourgroup's `LevelListMode` property to `'manual'`.

Drawing a Single Contour

MATLAB functions do not differentiate between a scalar and a one-element vector. So, if `v` is a one-element vector specifying a single contour at that level, `contour` interprets it as the number of contour lines, not the contour level. Consequently, `contour(Z,v)` behaves in the same manner as `contour(Z,n)`.

To display a single contour line, define `v` as a two-element vector with both elements equal to the desired contour level. For example, create a 3-D contour of the `peaks` function:

```
xrange = -3:.125:3;
yrange = xrange;
[X,Y] = meshgrid(xrange,yrange);
Z = peaks(X,Y);
contour3(X,Y,Z)
```

To display only one contour level at `Z = 1`, define `v` as :

```
v = [1 1]
contour3(X,Y,Z,v)
```

Example — Visualizing Contour Construction

You can think of a contour as the intersection of a 3-D surface with a horizontal plane. The intersection defines 0 or more *level lines* that trace contours. The level lines either form loops or terminate at the outer edges of the surface. Contour loops can intersect at saddle points, and therefore require special handling in their vicinity.

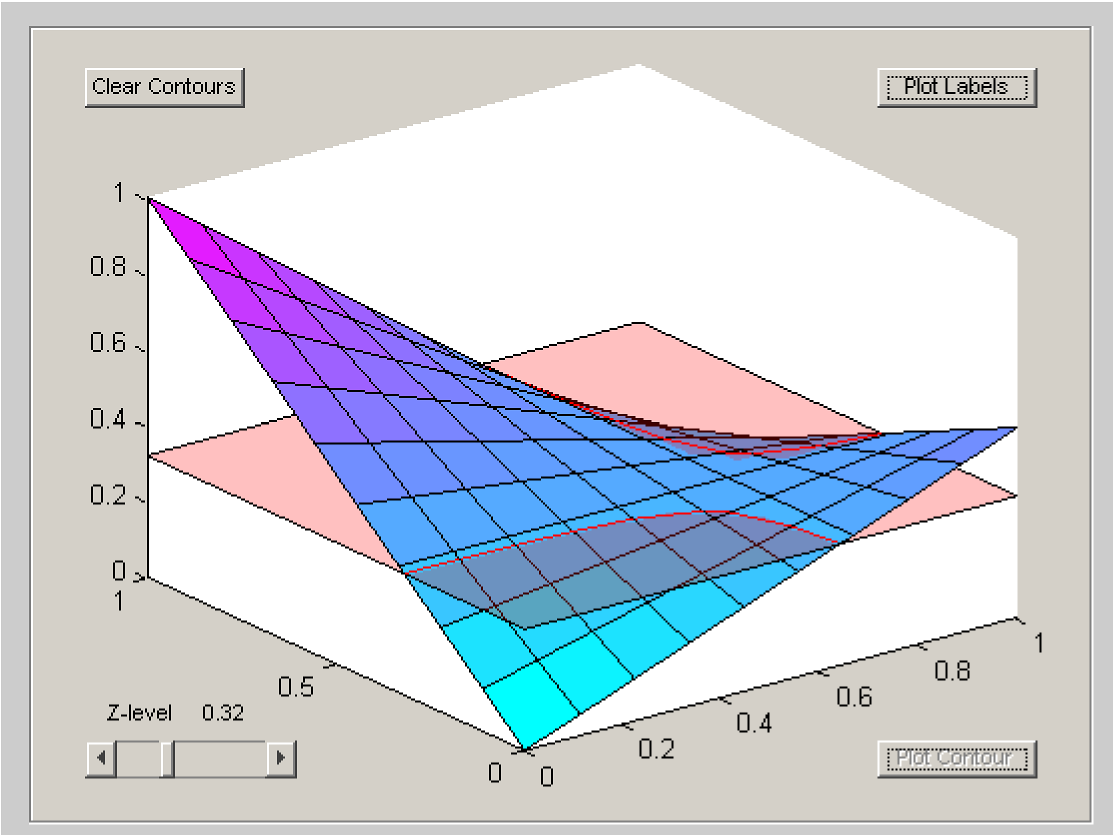
Run the following interactive code to visualize how contour lines are constructed. Use the slider to move the plane up or down through the range of *z* values, and click the **Plot Contour** button to draw a contour line that delineates where the plane slices through the surface. Click the **Plot Labels**

button to add a label to the contour you just plotted. Click **Clear Contours** to remove all the contours and labels.

```
% Create x, y, and z arrays for a parametric surface
[x y]=meshgrid(linspace(0,1,10),linspace(0,1,10));
z = .5*x + y - 1.5*x.*y;
% Display with the surface function in 3-D
fh = figure; colormap cool;
hpl = uipanel(fh,'Units','normalized','position',[.025 .025 .95 .95]);
s=surface('xdata',x,'ydata',y,'zdata',z,'cdata',z);
view(3);hold on;
% Display a second surface, a horixontal plane at z = 0
p=surface('xdata',[0 1;0 1],'ydata',[0 0; 1 1],...
          'zdata',[0 0; 0 0],'cdata',[0 0;0 0]);
set(p,'facealpha',.25,'facecolor','red'); % Make cut plane transparent
% Create a slider control for contour elevations
hs = uicontrol(hpl,'style','slider','min',0,'max',100,...
            'units','normalized','position',[.05 .05 .2 .05],...
            'sliderstep', [.01 .05]);
set(hs,'callback',...   % Tell the slider what it should do
    ['lvl=get(hs,''value'')/100;,' ...
     'set(p,''zdata'',[lvl lvl; lvl lvl]),' ...
     'set(hto,''string'',num2str(lvl)),' ...
     'set(hbc,''enable'',''on'')']);
 lvl = 0;      % Initialize the z-level of the cutting plane
% Create a label for the slider and a text box to show its value
hst = uicontrol(hpl,'Style','text', 'String','Z-level',...
                'units','normalized','Position',[.05 .10 .1 .05]);
hto = uicontrol(hpl,'Style','text', 'String','0',...
                'units','normalized','Position',[.13 .10 .1 .05]);
% Create a pushbutton control for drawing contours with CONTOUR3
hbc = uicontrol(hpl,'style','pushbutton','enable','off',...
    'string','Plot Contour',...
    'units','normalized','position',[.80 .05 .15 .05]);
set(hbc,'callback',['[C hc] = contour3(x,y,z, [lvl lvl],''r'');' ...
    'set(hbl,''enable'',''on''), set(hbe,''enable'',''on''),' ...
    'set(hbc,''enable'',''off'')']);
% Create a pushbutton control for labelling with CLABEL,
% which uses the "contour matrix" returned from CONTOUR3
hbl = uicontrol(hpl,'style','pushbutton','enable','off',...
```

```
    'string','Plot Labels',...
    'units','normalized','position',[.80 .90 .15 ,.05]);
set(hbl,'callback',['clabel(C, hc,''color'',''r'',' ...
   '''fontweight'',''bold'');' 'set(hbl,''enable'',''off''), '...
   'set(hbe,''enable'',''on'')']);
% Create a pushbutton to clear away the contours and labels
hbe = uicontrol(hpl,'style','pushbutton','enable','off',...
    'string','Clear Contours',...
    'units','normalized','position',[.05 .90 .15 .05]);
set(hbe,'callback',['delete(findall(gca,''color'',''r''));' ...
    'set(hbe,''enable'',''off'')']);
```

Here is what the figure and its controls look like with a contour plotted at the cut line.



See "The Contouring Algorithm" on page 5-70, below, for an explanation of how contour lines are computed.

Index Contours

You can index contours to visually emphasize certain contour levels. This technique, commonly used on topographic maps to highlight contours at set altitudes such as 25, 50, 75, ... meters above sea level, provides visual cues analogous to major ticks on a graph's axis. It is much easier to read a contour display that shows index contours because the heavier lines lessen the chance that your eye jumps between adjacent contours in scanning across the plot.

Example — Specifying Index Contours

The following code example highlights contours at elevations of –6, –5, –4, ... 7 for the output of the `peaks` function.

1 Generate a data matrix to contour:

```
z = peaks(100);
```

2 Compute 40 contour levels. Select contour levels so as to be round numbers; `zlevs` is the vector of contour levels to be plotted:

```
zmin = floor(min(z(:))); zmax = ceil(max(z(:)));
zinc = (zmax - zmin) / 40;
zlevs = zmin:zinc:zmax;
```

3 Specify the vertical distance between index contours; here it is unity, but it can be any modulus of values in `zlevs`.

```
zindex = 1;
```

4 Plot 2-D level lines with the `contour` function:

```
[c2,hc2] = contour(z,zlevs);
```

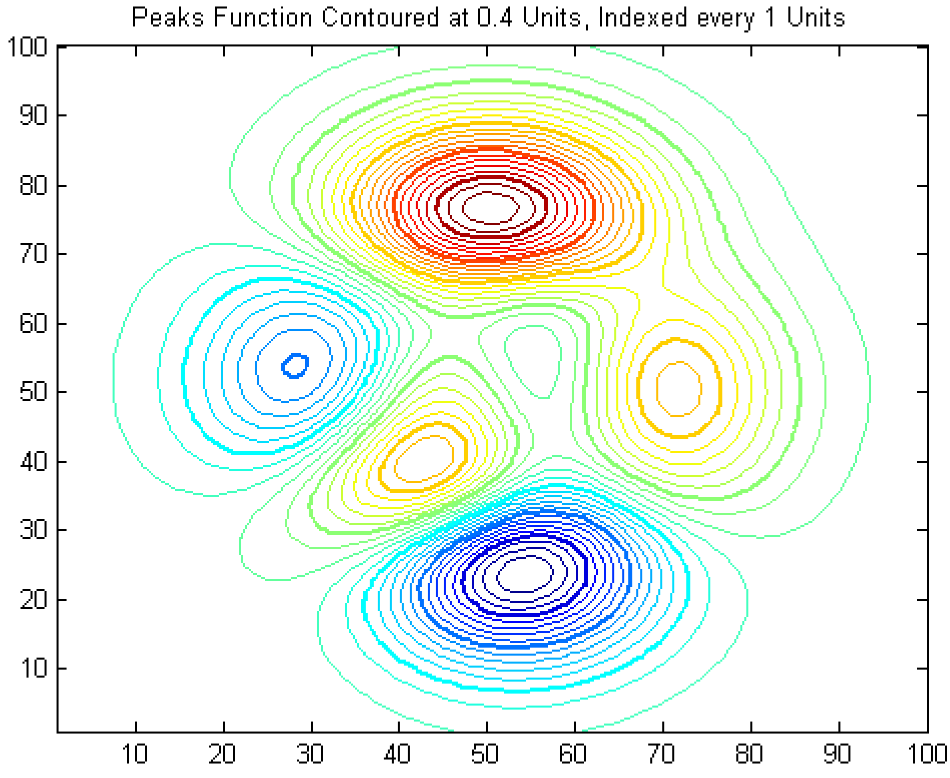
5 Create index contours by thickening level lines every `zindex` units:

```
nc = get(hc2,'Children');
for i = 1:length(nc)
   ud = get(nc(i),'UserData');
   if (mod(ud,zindex) == 0)
       set(nc(i),'LineWidth',2);
   end
end
```

A contour line thickens with each call to `set`.

6 Annotate to identify the contouring parameters used:

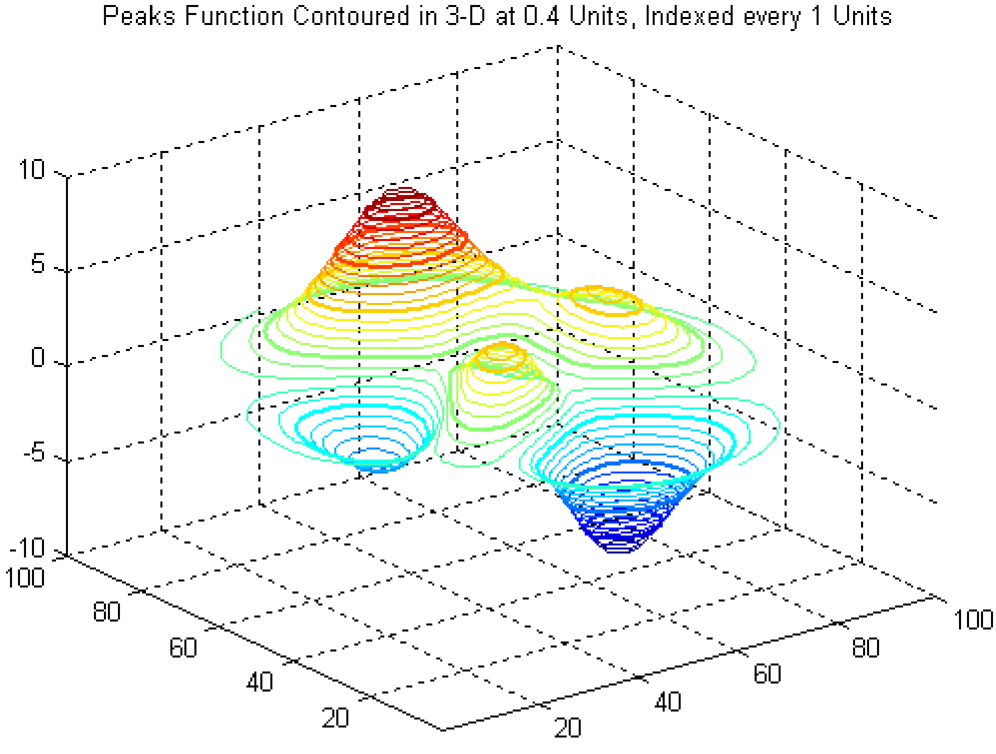
```
s = sprintf('%s %g %s %g %s', 'Peaks Function Contoured at', ...
    zinc, 'Units, Indexed every', zindex, 'Units');
title(s)
```



The loop of code in step 5 above works for `contour` but not for`contour3`, because `contour3` does not create `contourgroup` objects containing `Children`. To accomplish the same result with `contour3`, you must dereference the handle to the contours returned by `contour3` (`hc3`, below) differently, as follows:

```
figure;
[c3,hc3] = contour3(z,zlevs);
for i = 1:length(hc3)
   ud = get(hc3(i),'UserData');
   if (mod(ud,zindex) == 0)
       set(hc3(i),'LineWidth',2);
   end
end
```

```
s = sprintf('%s %g %s %g %s',...
    'Peaks Function Contoured in 3-D at', ...
    zinc, 'Units, Indexed every', zindex, 'Units');
title(s)
```



The Contouring Algorithm

The `contourc` function calculates the contour matrix for the other contour functions. It is a low-level function that is not called from the command line.

The contouring algorithm first determines which contour levels to draw. If you specified the input vector `v`, the elements of `v` are the contour level values, and `length(v)` determines the number of contour levels generated. If you do not specify `v`, the algorithm chooses no more than 20 contour levels that are divisible by 2 or 5.

The height matrix `Z` has associated `X` and `Y` matrices that locate each value in `Z` at the intersection of a row and a column, or these matrices are inferred when they are unspecified. The row and column widths can vary, but typically they are constant (i.e., `Z` is a regular grid).

Set the current level, `c`, equal to the lowest contour level to be plotted within the range `[min(Z) max(Z)]`. The contouring algorithm checks each edge of every square in the grid to see if `c` is between the two *z* values for the edge points. If so, a contour at that level crosses the edge, and a linear interpolation is performed:

```
t=(c-Z0)/(Z1-Z0)
```

`Z0` is the *z* value at one edge point, and `Z1` is the *z* value at the other edge point.

Start indexing a new contour line (`i = 1`) for level `c` by interpolating *x* and *y*:

```
cx(i) = X0+t*(X1-X0)
cy(i) = Y0+t*(Y1-Y0)
```

Walk around the edges of the square just entered; the contour exits at the next edge with *z* values that bracket `c`. Increment `i`, compute `t` for the edge, and then compute `cx(i)` and `cy(i)`, as above.

Mark the square as having been visited. Keep checking the edges of each square entered to determine the exit edge until the line`(cx,cy)` closes on its initial point or exits the grid. If the square being entered is already marked, the contour line closes there. Copy `cx`, `cy`, `c`, and `i` to the contour line data structure (the matrix returned by contouring functions, described shortly).

Reinitialize `cx`, `cy`, and `i`. Move to an unmarked square and test its edges for intersections; when you find one at level `c`, repeat the preceding operations. Any number of contour lines can exist for a given level.

Clear all the markers, increment the contour level, and repeat until `c` exceeds `max(Z)`.

Extra logic is needed for squares where a contour passes through all four edges (saddle points) to determine which pairs of edges to connect.

contour, contour3, and contourf return a two-row matrix that specifies all the contour lines:

```
C = [    value1    xdata(1)    xdata(2)...
         numv      ydata(1)    ydata(2)...]
```

The first row of the column that begins each definition of a contour line contains the contour value, as specified by v and used by clabel. Beneath that value is the number of (*x,y*) vertices in the contour line. Remaining columns contain the data for the (*x,y*) pairs. For example, the contour matrix calculated by C = contour(peaks(3)) is as follows.

Three vertices at v = −0.2

Three vertices at v = 0

Five vertices at v = 0.8

```
Columns 1 through 7
-0.2000    1.8165    2.0000    2.1835         0    1.0003    2.0000
 3.0000    1.0000    1.0367    1.0000    3.0000    1.0000    1.1998

Columns 8 through 14
 3.0000         0    1.0000    1.0359    1.0000    0.2000    1.6669
 1.0002    3.0000    2.9991    2.0000    1.0018    5.0000    3.0000

Columns 15 through 21
 1.2324    2.0000    2.8240    2.3331    0.4000    2.0000    2.6130
 2.0000    1.3629    2.0000    3.0000    5.0000    2.8530    2.0000

Columns 22 through 28
 2.0000    1.4290    2.0000    0.6000    2.0000    2.4020    2.0000
 1.5261    2.0000    2.8530    5.0000    2.5594    2.0000    1.6892

Columns 29 through 35
 1.6255    2.0000    0.8000    2.0000    2.1910    2.0000    1.8221
 2.0000    2.5594    5.0000    2.2657    2.0000    1.8524    2.0000

Column 36
 2.0000
 2.2657
```

The circled values begin each definition of a contour line.

Changing the Offset of a Contour

The surfc and meshc functions display contours beneath a surface or a mesh plot. These functions draw the contour plot at the axes' minimum *z*-axis limit.

To specify your own offset, change the `ZData` values of the contour lines. First, save the handles of the graphics objects created by `meshc` or `surfc`:

```
h = meshc(peaks(20));
```

The first handle belongs to the mesh or surface. The remaining handles belong to the contours you want to change. To raise the contour plane, increment the *z* coordinate of each contour line by some amount by resetting its `Zdata` value:

```
for i = 2:length(h);
    newz = get(h(i),'Zdata') + 5;
    set(h(i),'Zdata',newz)
end
```

Displaying Contours in Polar Coordinates

1 You can contour data defined in the polar coordinate system. As an example, set up a grid in polar coordinates and convert the coordinates to Cartesian coordinates.

```
[th,r] = meshgrid((0:5:360)*pi/180,0:.05:1);
[X,Y] = pol2cart(th,r);
```

2 Then generate the complex matrix `Z` on the interior of the unit circle.

```
Z = X+i*Y;
```

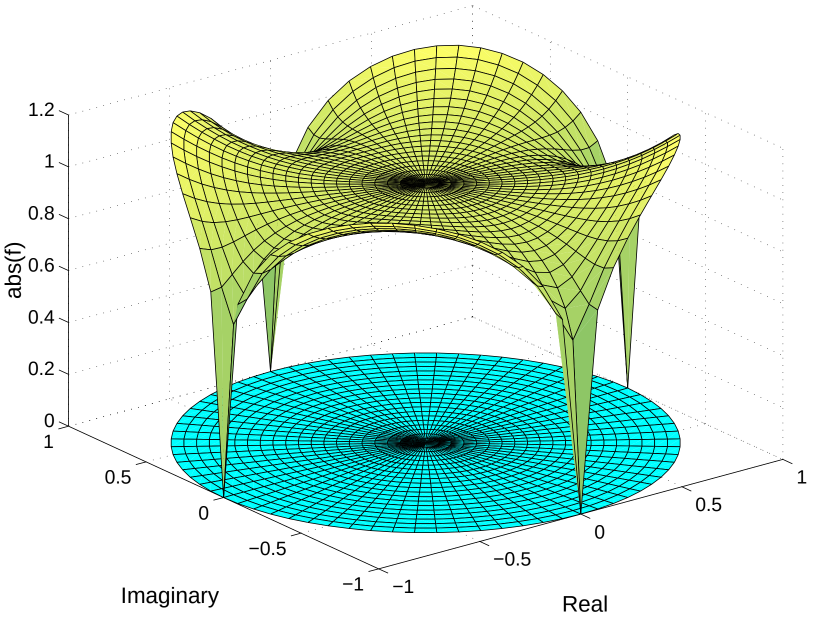
`X`, `Y`, and `Z` are points inside the circle.

3 Create and display a surface of the function $\sqrt[-4]{Z^4 - 1}$.

```
f = (Z.^4-1).^(1/4);
surf(X,Y,abs(f))
```

4 Display the unit circle beneath the surface and add labels to the graph:

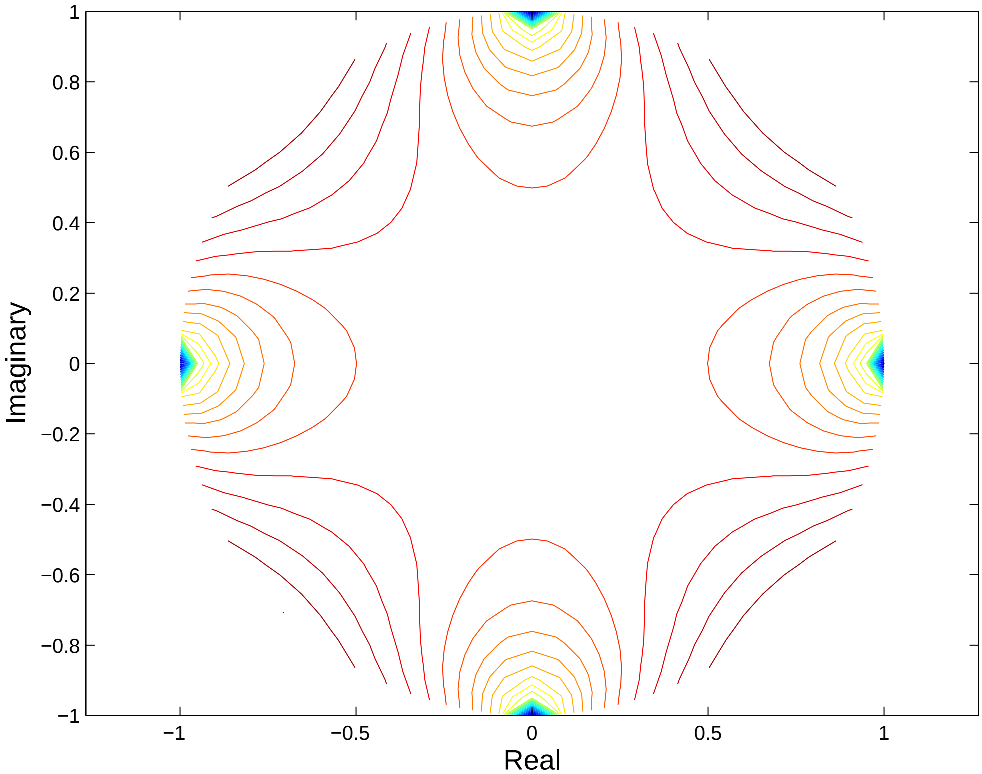
```
hold on
surf(X,Y,zeros(size(X)))
hold off
xlabel('Real','FontSize',14);
ylabel('Imaginary','FontSize',14);
zlabel('abs(f)','FontSize',14);
```



Contours in Cartesian Coordinates

These statements display a contour of the surface in Cartesian coordinates and label the *x*- and *y*-axis:

```
contour(X,Y,abs(f),30)
axis equal
xlabel('Real','FontSize',14);
ylabel('Imaginary','FontSize',14);
```



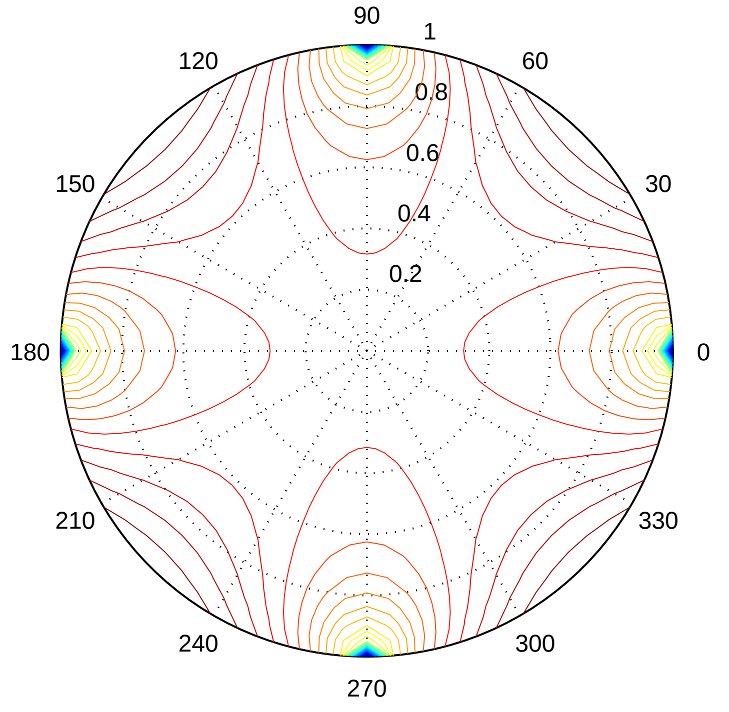
Contours on a Polar Axis

You can also display the contour within a polar axes. Create a polar axes using the `polar` function, and then delete the line specified with `polar`:

```
h = polar([0 2*pi], [0 1]);
delete(h)
```

With `hold on`, display the contour on the polar grid:

```
hold on
contour(X,Y,abs(f),30)
```



Preparing Data for Contouring

The various *contour* plotting functions, as well as the *mesh* and *surface* families of functions, accept 2-D matrices as inputs. For most applications, these input grids represent continuous functions of two variables or relatively continuous fields of data. In many applications, source data might consist of *z* values sampled over a two-dimensional domain in an irregular fashion, such as discrete spot elevations from GPS measurements (in the form of *x*, *y*, and *z* data vectors). To prepare such data for contour or mesh display, you need to *interpolate* it in some fashion.

There are several MATLAB methods for interpolating data into vectors, grids, and triangulated (Delaunay) tessellations. Input observations can be one-, two-, three- or higher-dimensional. By choosing and using these functions carefully you control parameters and constraints for interpolation to model your assumptions about the underlying nature of the raw data. Typically, you use the `interp2`, `meshgrid`, and `griddata` functions to interpolate *z* values for scattered *x-y* data points into a 2-D grid. See “Computational Geometry” in the MATLAB Mathematics documentation for discussion and examples of data interpolation using these and other functions.

If the surface you are contouring is “noisy,” contours depicting the surface exhibit jaggedness. When you analyze and explore such data, you can filter it to attentuate high-frequency variations. One way to do this is with a convolution (with `conv2` or `filter2`) filter, as the following example demonstrates:

Example — Smoothing a Matrix for Plotting Contours

The `conv2` and `filter` functions can remove high-frequency components from a matrix representing a continuous surface or field to make the underlying data easier to visualize.

1 Create a function of two variables and plot contour lines at a specified, fixed interval:

```
Z = peaks(100);
figure;
set (gcf,'position',[400,100,600,600], 'color','w')
subplot(2,2,1);
cl = [-7:1:10];          % Define contour levels for all plots
contour(Z, cl)
axis([0 100 0 100]); colormap autumn;
set(gca,'Xtick',[0 100],'Ytick',[0 100]);
title('Peaks Surface (underlying data)')
```

2 Add uniform random noise with mean of 0 to the surface and plot resulting contours. Irregularities in the contours tend to obscure the trend of the data:

```
ZN = Z + rand(100) - .5;
subplot (2,2,2)
```

```
contour(ZN, cl)
axis([0 100 0 100]);
set(gca,'Xtick',[0 100],'Ytick',[0 100]);
title('Peaks Surface (noise added)')
```

3 Specify a 3-by-3 convolution kernal, `F`, for smoothing the matrix and use the `conv2` function to attenuate high spatial frequencies in the surface data:

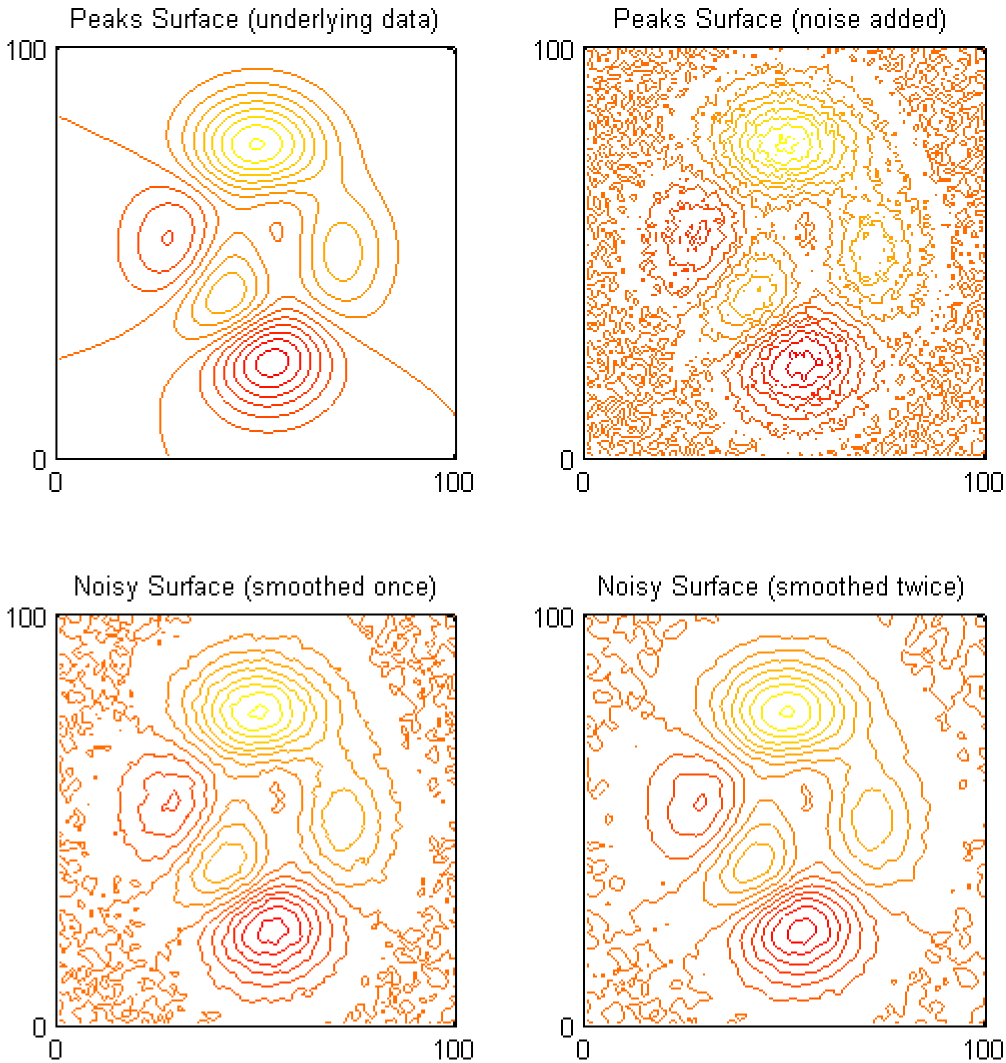
```
F = [.05 .1 .05; .1 .4 .1; .05 .1 .05];
ZC = conv2(ZN,F,'same');
```

4 Visually compare the smoothed surface to the original and the noisy ones:

```
subplot (2,2,3)
contour(ZC, cl)
axis([0 100 0 100]);
set(gca,'Xtick',[0 100],'Ytick',[0 100]);
title('Noisy Surface (smoothed once)')
```

5 Smooth the surface one more time using the same operator and compare (a larger or more uniform kernal can achieve this in one pass):

```
ZC2 = conv2(ZC,F,'same');
subplot (2,2,4)
contour(ZC2, cl)
axis([0 100 0 100]);
set(gca,'Xtick',[0 100],'Ytick',[0 100]);
title('Noisy Surface (smoothed twice)')
```


Peaks Surface (underlying data)
100
0
0
100
Peaks Surface (noise added)
100
0
0
100
Noisy Surface (smoothed once)
100
0
0
100
Noisy Surface (smoothed twice)
100
0
0
100

Interactive Plotting

Example — Selecting Plotting Points from the Screen

You can interact with graphs or generate *x-y* coordinates interactively. The `ginput` function enables you to use the mouse or the arrow keys to select points to plot. `ginput` returns the coordinates of the pointer's position, either the current position or the position when a mouse button or key is pressed. For more information see the `ginput` function. You can use it to pick points on a graph to return their *x* and *y* values for processing, to outline an area of interest, or to draw arbitrary shapes.

This example illustrates the use of `ginput` with the `spline` function to create a curve by interpolating in two dimensions.

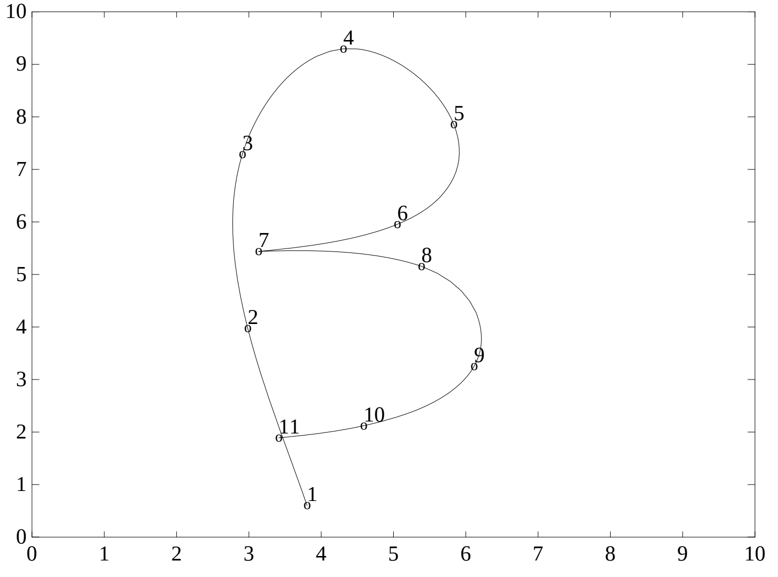
First, select a sequence of points, `[x,y]`, in the plane with `ginput`. Then pass two one-dimensional splines through the points, evaluating them with a spacing one-tenth of the original spacing:

```
axis([0 10 0 10])
hold on
% Initially, the list of points is empty.
xy = [];
n = 0;
% Loop, picking up the points.
disp('Left mouse button picks points.')
disp('Right mouse button picks last point.')
but = 1;
while but == 1
    [xi,yi,but] = ginput(1);
    plot(xi,yi,'ro')
    n = n+1;
    xy(:,n) = [xi;yi];
end
% Interpolate with a spline curve and finer spacing.
t = 1:n;
ts = 1: 0.1: n;
xys = spline(t,xy,ts);

% Plot the interpolated curve.
```

```
plot(xys(1,:),xys(2,:),'b-');
hold off
```

This plot shows some typical output:



Animation

In this section...

Ways to Animate Plots

You can create animated sequences with MATLAB graphics in three different ways:

- Save a number of different pictures and play them back as a movie.
- Continually erase and redraw the objects on the screen, making incremental changes with each redraw.
- Redefine the `XData`, `YData`, `ZData`, and/or `CData` plot object properties, optionally linking them to data sources (workspace variables) and updating the properties via calls to `refreshdata`.

Movies are better suited to situations where each frame is complex and cannot be redrawn rapidly. You create each movie frame in advance so the original drawing time is not important during playback, which is just a matter of blotting the frame to the screen. A movie is not rendered in real time; it is simply a playback of previously rendered frames.

The second technique, drawing, erasing, and then redrawing, makes use of different drawing modes supported by MATLAB graphics. These modes allow faster redrawing at the expense of some rendering accuracy, so you must consider which mode to select.

The third approach allows plots to be updated in data driven fashion and handles redrawing plots (if `drawnow` is called appropriately).

This section provides an example of each technique. To see more sophisticated demonstrations of these features, type `demo` at the MATLAB prompt and explore the animation demonstrations.

Movies

You can save any sequence of graphs and play the sequence back in a short *movie*. There are two steps to this process:

- Use `getframe` to generate each movie frame. Be sure that your computer is not in screen saver mode when you call `getframe`. In the event that you are using several virtual desktops, make sure that the desktop on which the MATLAB application is running is visible on your monitor.
- Use `movie` to run the movie a specified number of times at the specified rate.

Typically, you use `getframe` in a `for` loop to assemble the array of movie frames. `getframe` returns a structure having the following fields:

- `cdata` — Image data in a `uint8` matrix. The matrix has dimensions of height-by-width on indexed-color systems and height-by-width-by-3 on truecolor systems.
- `colormap` — The colormap in an *n*-by-3 matrix, where *n* is the number of colors. On truecolor systems, the `colormap` field is empty.

Example — Visualizing an FFT as a Movie

This example illustrates the use of movies to visualize the quantity `fft(eye(n))`, which is a complex *n*-by-*n* matrix whose elements are various powers of the nth root of unity, `exp(i*2*pi/n)`.

Creating the Movie

Create the movie in a `for` loop calling `getframe` to capture the graph. Since the `plot` command resets the axes properties, call `axis equal` within the loop before `getframe`:

```
for k = 1:16
 plot(fft(eye(k+16)))
 axis equal
 M(k) = getframe;
```

```
end
```

Running the Movie

After generating the movie, you can play it back any number of times. To play it back 30 times, type

```
movie(M,30)
```

You can readily generate and smoothly play back movies with a few dozen frames on most computers. Longer movies require large amounts of primary memory or a very effective virtual memory system.

Movies that Include the Entire Figure

To capture the contents of the entire figure window (for example, to include GUI components in the movie), specify the figure's handle as an argument to the `getframe` command. For example, suppose you want to add a slider to indicate the value of `k` in the previous example. The following code introduces a slider on the left side of the figure.

```
h = uicontrol('style','slider','position',...
 [10 50 20 300],'Min',1,'Max',16,'Value',1)
for k = 1:16
 plot(fft(eye(k+16)))
 axis equal
 set(h,'Value',k)
 M(k) = getframe(gcf);
end
```

In this example, the movie frame contains the entire figure. To play the movie so that it looks like the original figure, make the playback axes fill the figure window.

```
clf
axes('Position',[0 0 1 1])
movie(M,30)
```

Updating Plot Object Axis and Color Data

When you create a graph the MATLAB figure stores copies of

- Data it needs to define *x*, *y*, and *z*
- Color values it depicts in the plot object itself (e.g., lineseries, barseries, surfaceplot, etc.)

If the variables that these values represent are removed or changed, the copies of them in plot object are unaffected. However, you can update these copies (the properties `XData`, `YData`, `ZData`, and `CData`) at any time; when you do, the graph changes to reflect the updates.

Consequently, you can animate graphs by changing axis data. Do one of the following:

- Explicitly provide new axis data to a graph by calling `set` directly, e.g.,

  ```
  set(obj_handle,'YData',[3 5 8 6 7 0])
  ```

- Implicitly update a graph when a workspace variable changes value, by first calling `set` to define a *data source* for an axis, e.g.,

  ```
  set(obj_handle,'YDataSource','varname')
  ```

  and then call `refreshdata` after you or your code updates `varname`.

`obj_handle` is a handle to the plot object you want to update or animate. All graph objects have data source properties (at least an `XDatasource` and a `YDatasource`; some also have a `ZDatasource` and a `CDatasource`) that by default are empty (axis data has no connection to workspace variables).

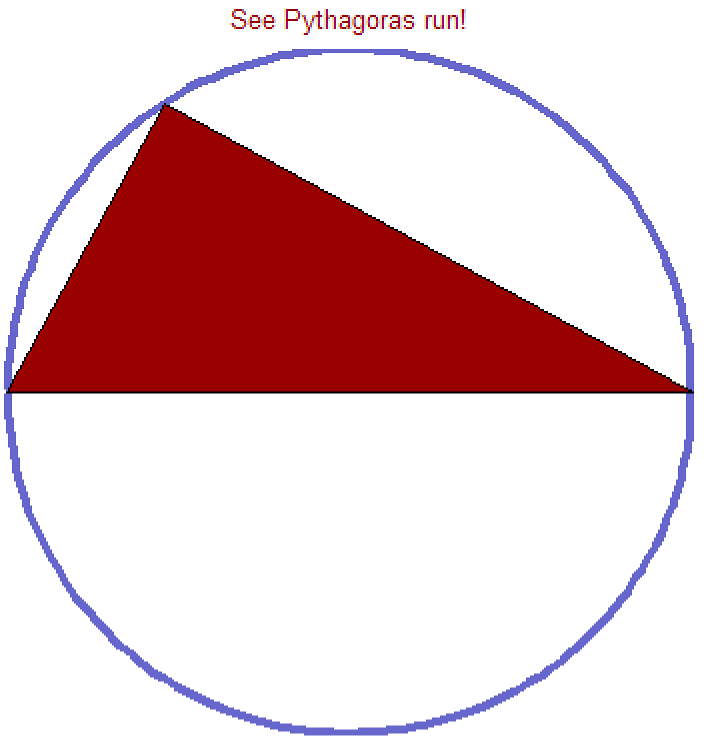
You make a persistent connection between axis data and a variable using `set`, as described earlier and illustrated by the following example. When you call `refreshdata`, the workspace data replaces the axis data and the program redraws the graph.

Calling `refreshdata` only causes a graph to be redrawn if any of its declared data sources have changed. It updates all axes at the same time, and updates selected plot objects from a calling function's workspace or the base workspace.

As an example, this script calls `refreshdata` to animate an area graph of the Pythagorean theorem:

```
c = -pi:.04:pi;
cx = cos(c);
cy = -sin(c);
figure('color','white');
axis off, axis equal
line(cx, cy, 'color', [.4 .4 .8],'LineWidth',3);
title('See Pythagoras Run!','Color',[.6 0 0])
hold on
x = [-1 0 1 -1];
y =   [0 0 0 0];
ht = area(x,y,'facecolor',[.6 0 0])
set(ht,'XDataSource','x')
set(ht,'YDataSource','y')
for j = 1:length(c)
    x(2) = cx(j);
    y(2) = cy(j);
    refreshdata
    drawnow
end
```

The script needs `drawnow` to display the results at each iteration. When you call `refreshdata` from the command line or manually set the `XData`, `YData`, `ZData`, or `CData` of a graph, the plot redraws automatically. One frame from the animation looks like this.



For more information, see the `refreshdata` reference page.

To program the same animation without using `refreshdata`, the code becomes

```
c = -pi:.04:pi;
cx = cos(c);
cy = -sin(c);
figure('color','white');
axis off, axis equal
line(cx, cy, 'color', [.4 .4 .8],'LineWidth',3);
title('See Pythagoras run!','Color',[.6 0 0])
hold on
x = [-1 0 1 -1];
y =   [0 0 0 0];
ht = area(x,y,'facecolor',[.6 0 0]);
for j = 1:length(c)
```

```
        x(2) = cx(j);
        y(2) = cy(j);
        set(ht,'XData',x)
        set(ht,'YData',y)
        drawnow
    end
```

This code directly assigns plot axis data. Because there is less evaluation going on, it runs visibly faster. The advantage to using `refreshdata` is that it makes it easier for a program to keep plots in sync when workspace data changes.

Updating Graphs with linkdata Versus refreshdata

The `linkdata` function, which you can activate and deactivate with the **Data Linking** tool  on the figure toolbar, is another way to update a graph when any of its data sources change. When it is turned on, the tool updates axis data continuously and automatically, without calling `refreshdata`. However, *data linking is not intended to animate plots*; rather, its purpose is to keep different plots in sync and to extend the capabilities of Data Brushing mode, in which you manually highlight observations of interest on a plot. When you use data brushing and data linking together, highlighting observations on one plot causes them to highlight on other plots which display `XData`, `YData`, or `ZData` from the same data sources.

Data linking is not useful for animation because it does not update plots immediately when data source value changes. Instead, it batches updates at roughly half-second intervals to reduce the communications involved in keeping plots and workspace variables synchronized. Therefore, you should not be using data linking at the same time you animate graphs using either of the techniques described above.

For more information on data linking and data brushing, see “Marking Up Graphs with Data Brushing” and “Making Graphs Responsive with Data Linking” in the MATLAB Data Analysis documentation.

Displaying Bit-Mapped Images

Working with Images in MATLAB Graphics

In this section...

What Is Image Data?

The basic MATLAB data structure is the *array*, an ordered set of real or complex elements. An array is naturally suited to the representation of *images*, real-valued, ordered sets of color or intensity data. (An array is suited for complex-valued images.)

In the MATLAB workspace, most images are represented as two-dimensional arrays (matrices), in which each element of the matrix corresponds to a single pixel in the displayed image. For example, an image composed of 200 rows and 300 columns of different colored dots stored as a 200-by-300 matrix. Some images, such as RGB, require a three-dimensional array, where the first plane in the third dimension represents the red pixel intensities, the second plane represents the green pixel intensities, and the third plane represents the blue pixel intensities.

This convention makes working with graphics file format images similar to working with any other type of matrix data. For example, you can select a single pixel from an image matrix using normal matrix subscripting:

```
I(2,15)
```

This command returns the value of the pixel at row 2, column 15 of the image `I`.

The following sections describe the different data and image types, and give details about how to read, write, work with, and display graphics images; how to alter the display properties and aspect ratio of an image during display; how to print an image; and how to convert the data type or graphics format of an image.

Data Types

MATLAB math supports three different numeric classes for image display:

- double-precision floating-point (`double`)
- 16-bit unsigned integer (`uint16`)
- 8-bit unsigned integer (`uint8`)

The image display commands interpret data values differently depending on the numeric class the data is stored in. “Working with 8-Bit and 16-Bit Images” on page 6-10 includes details on the inner workings of the storage for 8- and 16-bit images.

By default, most data occupy arrays of class `double`. The data in these arrays is stored as double-precision (64-bit) floating-point numbers. All MATLAB functions and capabilities work with these arrays.

For images stored in one of the graphics file formats supported by MATLAB functions, however, this data representation is not always ideal. The number of pixels in such an image can be very large; for example, a 1000-by-1000 image has a million pixels. Since at least one array element represents each pixel , this image requires about 8 megabytes of memory if it is stored as class `double`.

To reduce memory requirements, you can store image data in arrays of class `uint8` and `uint16`. The data in these arrays is stored as 8-bit or 16-bit unsigned integers. These arrays require one-eighth or one-fourth as much memory as data in `double` arrays.

Bit Depth

MATLAB input functions read the most commonly used bit depths (bits per pixel) of any of the supported graphics file formats. When the data is in memory, it can be stored as `uint8`, `uint16`, or `double`. For details on which bit depths are appropriate for each supported format, see `imread` and `imwrite`.

Supported Image Formats

MATLAB commands read, write, and display several types of graphics file formats for images. As with MATLAB generated images, once a graphics

file format image is displayed, it becomes a Handle Graphics image object. MATLAB supports the following graphics file formats, along with others:

- BMP (Microsoft Windows Bitmap)
- GIF (Graphics Interchange Files)
- HDF (Hierarchical Data Format)
- JPEG (Joint Photographic Experts Group)
- PCX (Paintbrush)
- PNG (Portable Network Graphics)
- TIFF (Tagged Image File Format)
- XWD (X Window Dump)

For more information about the bit depths and image types supported for these formats, see `imread` and `imwrite`.

Functions for Reading, Writing, and Displaying Images

Images are essentially two-dimensional matrices, so many MATLAB functions can operate on and display images. The following table lists the most useful ones. The sections that follow describe these functions in more detail.

Function	Purpose	Function Group
`axis`	Plot axis scaling and appearance.	Display
`image`	Display image (create image object).	Display
`imagesc`	Scale data and display as image.	Display
`imread`	Read image from graphics file.	File I/O
`imwrite`	Write image to graphics file.	File I/O
`imfinfo`	Get image information from graphics file.	Utility
`ind2rgb`	Convert indexed image to RGB image.	Utility

Image Types

In this section...

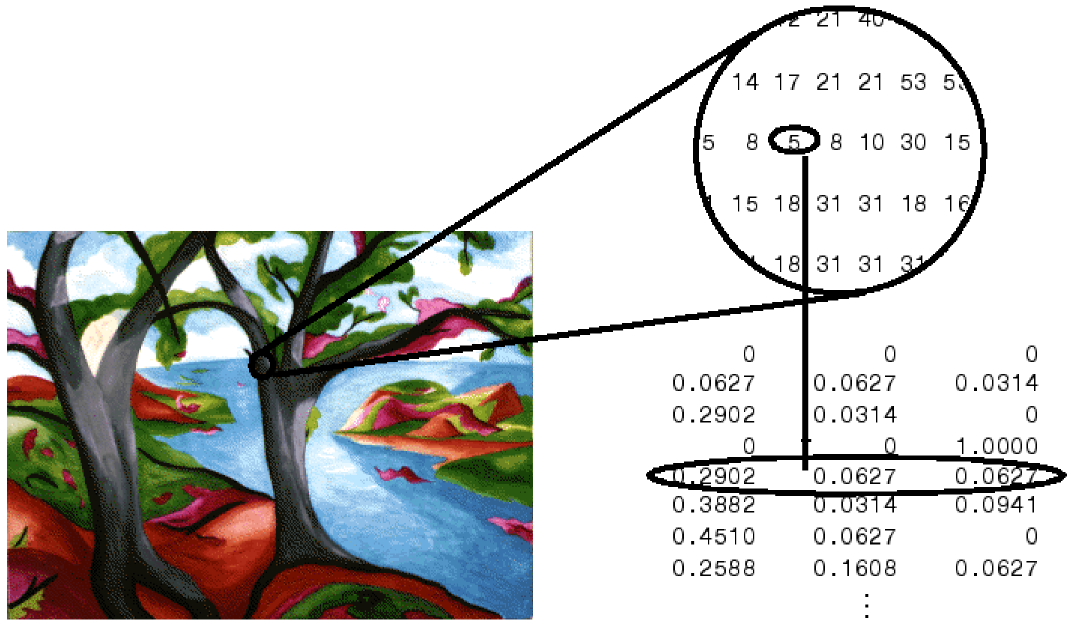
Indexed Images

An indexed image consists of a data matrix, `X`, and a colormap matrix, `map`. `map` is an *m*-by-3 array of class `double` containing floating-point values in the range [0, 1]. Each row of `map` specifies the red, green, and blue components of a single color. An indexed image uses "direct mapping" of pixel values to colormap values. The color of each image pixel is determined by using the corresponding value of `X` as an index into `map`. Values of `X` therefore must be integers. The value 1 points to the first row in `map`, the value 2 points to the second row, and so on. Display an indexed image with the statements

```
image(X); colormap(map)
```

A colormap is often stored with an indexed image and is automatically loaded with the image when you use the `imread` function. However, you are not limited to using the default colormap—use any colormap that you choose. The description for the property `CDataMapping` describes how to alter the type of mapping used.

The next figure illustrates the structure of an indexed image. The pixels in the image are represented by integers, which are pointers (indices) to color values stored in the colormap.



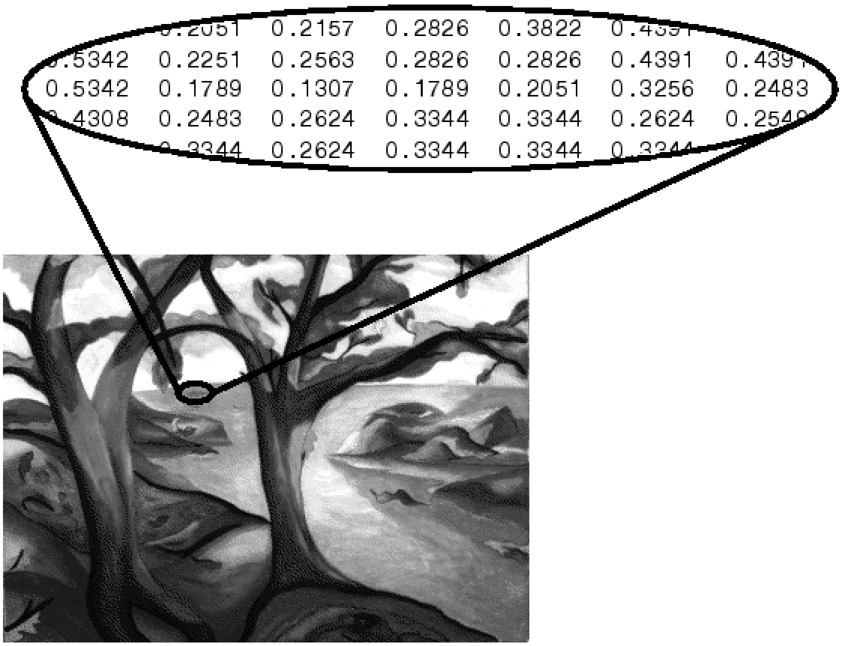
The relationship between the values in the image matrix and the colormap depends on the class of the image matrix. If the image matrix is of class `double`, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. If the image matrix is of class `uint8` or `uint16`, there is an offset—the value 0 points to the first row in the colormap, the value 1 points to the second row, and so on. The offset is also used in graphics file formats to maximize the number of colors that can be supported. In the preceding image, the image matrix is of class `double`. Because there is no offset, the value 5 points to the fifth row of the colormap.

Note When using the painters renderer on the Windows platform, you should only use 256 colors when attempting to display an indexed image. Larger colormaps can lead to unexpected colors because the painters algorithm uses the Windows 256 color palette, which graphics drivers and graphics hardware are known to handle differently. To work around this issue, use the `Zbuffer` or `OpenGL` renderer, as appropriate. For more information regarding graphics renderers in MATLAB, see `Technical Note 1201: The Technical Support Guide to Graphics Rendering and Troubleshooting`.

Intensity Images

An intensity image is a data matrix, `I`, whose values represent intensities within some range. An intensity image is represented as a single matrix, with each element of the matrix corresponding to one image pixel. The matrix can be of class `double`, `uint8`, or `uint16`. While intensity images are rarely saved with a colormap, a colormap is still used to display them. In essence, handles intensity images are treated as indexed images.

This figure depicts an intensity image of class `double`.



To display an intensity image, use the `imagesc` ("image scale") function, which enables you to set the range of intensity values. `imagesc` scales the image data to use the full colormap. Use the two-input form of `imagesc` to display an intensity image, for example:

```
imagesc(I,[0 1]); colormap(gray);
```

The second input argument to `imagesc` specifies the desired intensity range. The `imagesc` function displays `I` by mapping the first value in the range

(usually 0) to the first colormap entry, and the second value (usually 1) to the last colormap entry. Values in between are linearly distributed throughout the remaining colormap colors.

Although it is conventional to display intensity images using a grayscale colormap, it is possible to use other colormaps. For example, the following statements display the intensity image I in shades of blue and green:

```
imagesc(I,[0 1]); colormap(winter);
```

To display a matrix A with an arbitrary range of values as an intensity image, use the single-argument form of imagesc. With one input argument, imagesc maps the minimum value of the data matrix to the first colormap entry, and maps the maximum value to the last colormap entry. For example, these two lines are equivalent:

```
imagesc(A); colormap(gray)
imagesc(A,[min(A(:)) max(A(:))]); colormap(gray)
```

RGB (Truecolor) Images

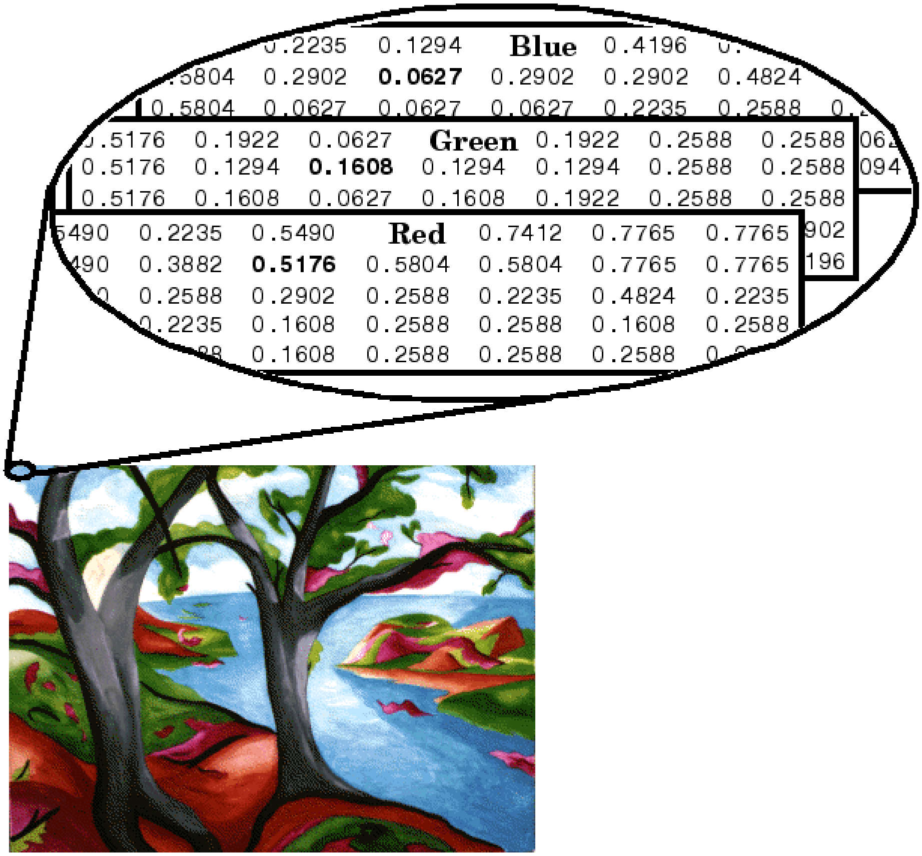
An RGB image, sometimes referred to as a *truecolor* image, is stored as an *m*-by-*n*-by-3 data array that defines red, green, and blue color components for each individual pixel. RGB images do not use a palette. The color of each pixel is determined by the combination of the red, green, and blue intensities stored in each color plane at the pixel's location. Graphics file formats store RGB images as 24-bit images, where the red, green, and blue components are 8 bits each. This yields a potential of 16 million colors. The precision with which a real-life image can be replicated has led to the nickname "truecolor image."

An RGB MATLAB array can be of class double, uint8, or uint16. In an RGB array of class double, each color component is a value between 0 and 1. A pixel whose color components are (0,0,0) is displayed as black, and a pixel whose color components are (1,1,1) is displayed as white. The three color components for each pixel are stored along the third dimension of the data array. For example, the red, green, and blue color components of the pixel (10,5) are stored in RGB(10,5,1), RGB(10,5,2), and RGB(10,5,3), respectively.

To display the truecolor image RGB, use the image function:

```
image(RGB)
```

The next figure shows an RGB image of class `double`.



To determine the color of the pixel at (2,3), look at the RGB triplet stored in (2,3,1:3). Suppose (2,3,1) contains the value `0.5176`, (2,3,2) contains `0.1608`, and (2,3,3) contains `0.0627`. The color for the pixel at (2,3) is

```
0.5176 0.1608 0.0627
```

Working with 8-Bit and 16-Bit Images

In this section...

8-Bit and 16-Bit Indexed Images

Double-precision (64-bit) floating-point numbers are the default MATLAB representation for numeric data. However, to reduce memory requirements for working with images, you can store images as 8-bit or 16-bit unsigned integers using the numeric classes `uint8` or `uint16`, respectively. An image whose data matrix has class `uint8` is called an 8-bit image; an image whose data matrix has class `uint16` is called a 16-bit image.

The `image` function can display 8- or 16-bit images directly without converting them to double precision. However, `image` interprets matrix values slightly differently when the image matrix is `uint8` or `uint16`. The specific interpretation depends on the image type.

If the class of `X` is `uint8` or `uint16`, its values are offset by 1 before being used as colormap indices. The value 0 points to the first row of the colormap, the value 1 points to the second row, and so on. The `image` command automatically supplies the proper offset, so the display method is the same whether `X` is `double`, `uint8`, or `uint16`:

```
image(X); colormap(map);
```

The colormap index offset for `uint8` and `uint16` data is intended to support standard graphics file formats, which typically store image data in indexed form with a 256-entry colormap. The offset allows you to manipulate and

display images of this form using the more memory-efficient uint8 and uint16 arrays.

Because of the offset, you must add 1 to convert a uint8 or uint16 indexed image to double. For example:

```
X64 = double(X8) + 1;
 or
X64 = double(X16) + 1;
```

Conversely, subtract 1 to convert a double indexed image to uint8 or uint16:

```
X8 = uint8(X64 - 1);
 or
X16 = uint16(X64 - 1);
```

8-Bit and 16-Bit Intensity Images

The range of double image arrays is usually [0, 1], but the range of 8-bit intensity images is usually [0, 255] and the range of 16-bit intensity images is usually [0, 65535]. Use the following command to display an 8-bit intensity image with a grayscale colormap:

```
imagesc(I,[0 255]); colormap(gray);
```

To convert an intensity image from double to uint16, first multiply by 65535:

```
I16 = uint16(round(I64*65535));
```

Conversely, divide by 65535 after converting a uint16 intensity image to double:

```
I64 = double(I16)/65535;
```

8-Bit and 16-Bit RGB Images

The color components of an 8-bit RGB image are integers in the range [0, 255] rather than floating-point values in the range [0, 1]. A pixel whose color components are (255,255,255) is displayed as white. The image command displays an RGB image correctly whether its class is double, uint8, or uint16:

```
image(RGB);
```

To convert an RGB image from double to uint8, first multiply by 255:

```
RGB8 = uint8(round(RGB64*255));
```

Conversely, divide by 255 after converting a uint8 RGB image to double:

```
RGB64 = double(RGB8)/255
```

To convert an RGB image from double to uint16, first multiply by 65535:

```
RGB16 = uint16(round(RGB64*65535));
```

Conversely, divide by 65535 after converting a uint16 RGB image to double:

```
RGB64 = double(RGB16)/65535;
```

Mathematical Operations Support for uint8 and uint16

To use the following MATLAB functions with uint8 and uint16 data, first convert the data to type double:

- conv2
- convn
- fft2
- fftn

For example, if X is a uint8 image, cast the data to type double:

```
fft(double(X))
```

In these cases, the output is always double.

The sum function returns results in the same type as its input, but provides an option to use double precision for calculations.

MATLAB Integer Mathematics

See "Arithmetic Operations on Integer Classes" for more information on how mathematical functions work with data types that are not doubles.

Most Image Processing Toolbox™ functions accept `uint8` and `uint16` input. If you plan to do sophisticated image processing on `uint8` or `uint16` data, consider including that toolbox in your MATLAB computing environment.

Other 8-Bit and 16-Bit Array Support

You can perform several other operations on `uint8` and `uint16` arrays, including:

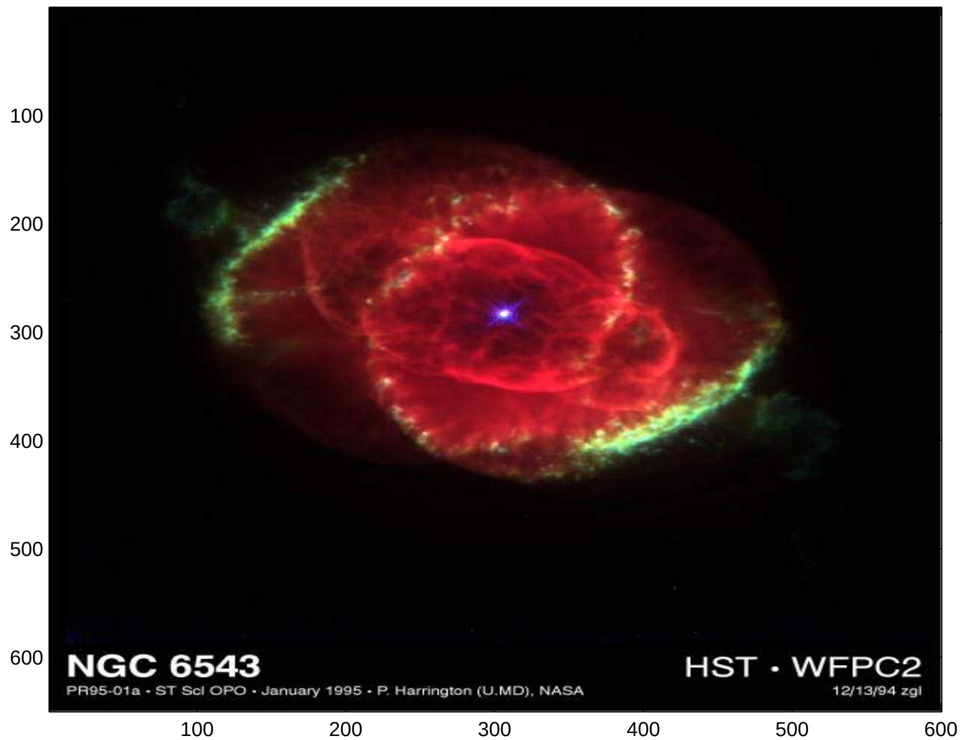
- Reshaping, reordering, and concatenating arrays using the functions `reshape`, `cat`, `permute`, and the `[]` and `'` operators
- Saving and loading `uint8` and `uint16` arrays in MAT-files using `save` and `load`. (Remember that if you are loading or saving a graphics file format image, you must use the commands `imread` and `imwrite` instead.)
- Locating the indices of nonzero elements in `uint8` and `uint16` arrays using `find`. However, the returned array is always of class `double`.
- Relational operators

Converting an 8-Bit RGB Image to Grayscale

You can perform arithmetic operations on integer data, which enables you to convert image types without first converting the numeric class of the image data.

This example reads an 8-bit RGB image into a MATLAB variable and converts it to a grayscale image:

```
rgb_img = imread('ngc6543a.jpg'); % Load the image
image(rgb_img) % Display the RGB image
```



Calculate the monochrome luminance by combining the RGB values according to the NTSC standard, which applies coefficients related to the eye's sensitivity to RGB colors:

```
I = .2989*rgb_img(:,:,1)...
 +.5870*rgb_img(:,:,2)...
 +.1140*rgb_img(:,:,3);
```

`I` is an intensity image with integer values ranging from a minimum of zero:

```
min(I(:))
ans =
```

```
    0
```

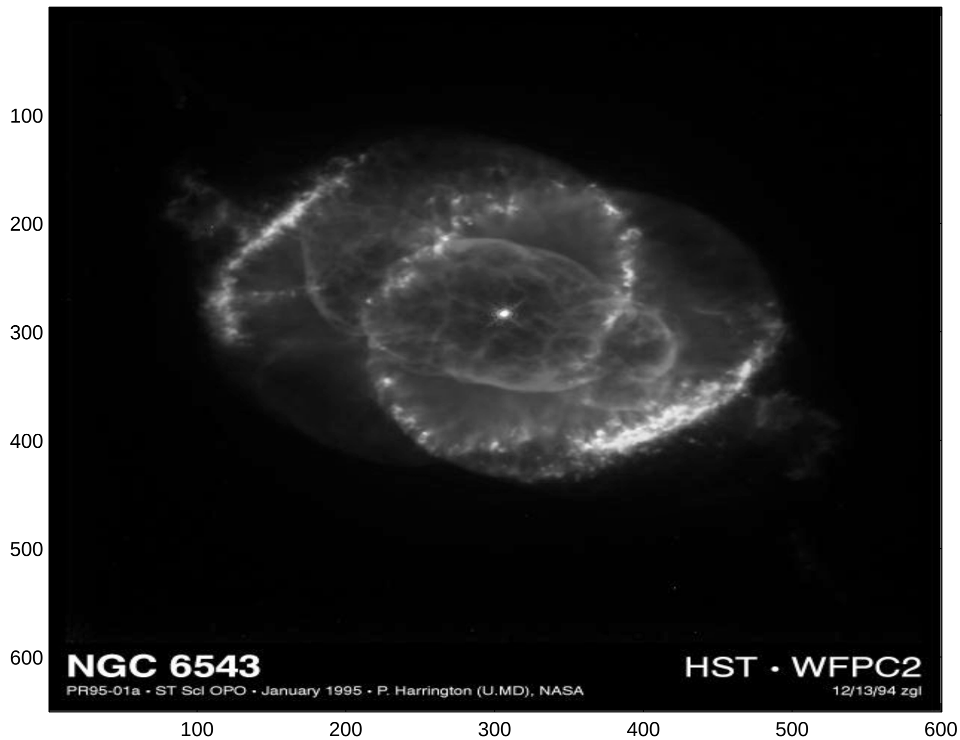
to a maximum of 255:

```
max(I(:))
ans =
  255
```

To display the image, use a grayscale colormap with 256 values. This avoids the need to scale the data-to-color mapping, which is required if you use a colormap of a different size. Use the `imagesc` function in cases where the colormap does not contain one entry for each data value.

Now display the image in a new figure using the gray colormap:

```
figure; colormap(gray(256)); image(I)
```



Related Information

Other colormaps with a range of colors that vary continuously from dark to light can produce usable images. For example, try `colormap(summer(256))` for a classic oscilloscope look. See `colormap` for more choices.

The `brighten` function enables you to increase or decrease the color intensities in a colormap to compensate for computer display differences or to enhance the visibility of faint or bright regions of the image (at the expense of the opposite end of the range).

Summary of Image Types and Numeric Classes

This table summarizes how data matrix elements are interpreted as pixel colors, depending on the image type and data class.

Image Type	double Data	uint8 or uint16 Data
Indexed	Image is an m-by-n array of integers in the range [1, p]. Colormap is a p-by-3 array of floating-point values in the range [0, 1].	Image is an m-by-n array of integers in the range [0, p –1]. Colormap is a p-by-3 array of floating-point values in the range [0, 1].
Intensity	Image is an m-by-n array of floating-point values that are linearly scaled to produce colormap indices. The typical range of values is [0, 1]. Colormap is a p-by-3 array of floating-point values in the range [0, 1] and is typically grayscale.	Image is an m-by-n array of integers that are linearly scaled to produce colormap indices. The typical range of values is [0, 255] or [0, 65535]. Colormap is a p-by-3 array of floating-point values in the range [0, 1] and is typically grayscale.
RGB (Truecolor)	Image is an m-by-n-by-3 array of floating-point values in the range [0, 1].	Image is an m-by-n-by-3 array of integers in the range [0, 255] or [0, 65535].

Reading, Writing, and Querying Graphics Image Files

In this section...

Working with Image Formats

In its native form, a graphics file format image is not stored as a MATLAB matrix, or even necessarily as a matrix. Most graphics files begin with a header containing format-specific information tags, and continue with bitmap data that can be read as a continuous stream. For this reason, you cannot use the standard MATLAB I/O commands `load` and `save` to read and write a graphics file format image.

Call special MATLAB functions to read and write image data from graphics file formats:

- To read a graphics file format image use `imread`.
- To write a graphics file format image, use `imwrite`.
- To obtain information about the nature of a graphics file format image, use `imfinfo`.

This table gives a clearer picture of which MATLAB commands should be used with which image types.

Procedure	Functions to Use
Load or save a matrix as a MAT-file.	`load` `save`
Load or save graphics file format image, e.g., BMP, TIFF.	`imread` `imwrite`

Procedure	Functions to Use
Display any image loaded into the MATLAB workspace.	image imagesc
Utilities	imfinfo ind2rgb

Reading a Graphics Image

The `imread` function reads an image from any supported graphics image file in any of the supported bit depths. Most of the images that you read are 8-bit. When these are read into memory, they are stored as class `uint8`. The main exception to this rule is MATLAB support for 16-bit data for PNG and TIFF images; if you read a 16-bit PNG or TIFF image, it is stored as class `uint16`.

Note For indexed images, `imread` always reads the colormap into an array of class `double`, even though the image array itself can be of class `uint8` or `uint16`.

The following commands read the image `ngc6543a.jpg` into the workspace variable `RGB` and then displays the image using the `image` function:

```
RGB = imread('ngc6543a.jpg');
image(RGB)
```

You can write (save) image data using the `imwrite` function. The statements

```
load clown % An image that is included with MATLAB
imwrite(X,map,'clown.bmp')
```

create a BMP file containing the clown image.

Writing a Graphics Image

When you save an image using `imwrite`, the default behavior is to automatically reduce the bit depth to `uint8`. Many of the images used in MATLAB are 8-bit, and most graphics file format images do not require

double-precision data. One exception to the rule for saving the image data as uint8 is that PNG and TIFF images can be saved as uint16. Because these two formats support 16-bit data, you can override the MATLAB default behavior by specifying uint16 as the data type for imwrite. The following example shows writing a 16-bit PNG file using imwrite.

```
imwrite(I,'clown.png','BitDepth',16);
```

Subsetting a Graphics Image (Cropping)

Sometimes you want to work with only a portion of an image file or you want to break it up into subsections. Specify the pixel coordinates of the rectangular subsection you want to work with and save it to a file from the command line. If you do not know the coordinates of the corner points of the subsection, choose them interactively, as the following example shows:

```
% Read demo RGB image from graphics file.
im = imread('street2.jpg');

% Display image with true aspect ratio
image(im); axis image

% Use ginput to select corner points of a rectangular
% region by pointing and clicking the mouse twice
p = ginput(2);

% Get the x and y corner coordinates as integers
sp(1) = min(floor(p(1)), floor(p(2))); %xmin
sp(2) = min(floor(p(3)), floor(p(4))); %ymin
sp(3) = max(ceil(p(1)), ceil(p(2)));   %xmax
sp(4) = max(ceil(p(3)), ceil(p(4)));   %ymax

% Index into the original image to create the new image
MM = im(sp(2):sp(4), sp(1): sp(3),:);

% Display the subsetted image with appropriate axis ratio
figure; image(MM); axis image

% Write image to graphics file.
imwrite(MM,'street2_cropped.tif')
```

If you know what the image corner coordinates should be, you can manually define `sp` in the preceding example rather than using `ginput`.

You can also display a “rubber band box” as you interact with the image to subset it. See the code example for `rbbox` for details. For further information, see the documentation for the `ginput` and `image` functions.

Obtaining Information About Graphics Files

The `imfinfo` function enables you to obtain information about graphics files in any of the standard formats listed earlier. The information you obtain depends on the type of file, but it always includes at least the following:

- Name of the file, including the directory path if the file is not in the current directory
- File format
- Version number of the file format
- File modification date
- File size in bytes
- Image width in pixels
- Image height in pixels
- Number of bits per pixel
- Image type: RGB (truecolor), intensity (grayscale), or indexed

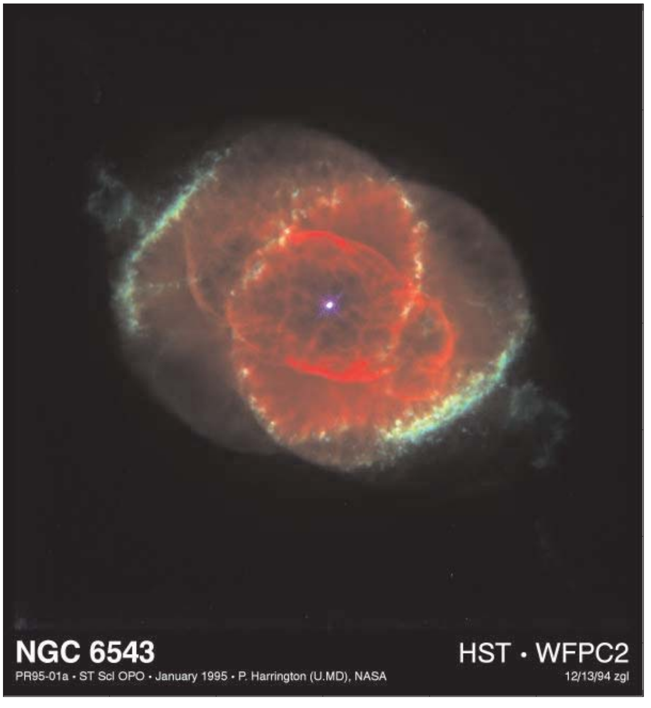
Displaying Graphics Images

In this section...

Image Types and Display Methods

To display a graphics file image, use either image or imagesc. For example, assuming RGB is an image,

```
figure('Position',[100 100 size(RGB,2) size(RGB,1)]);
image(RGB); set(gca,'Position',[0 0 1 1])
```



Note This image was created with the support of the Space Telescope Science Institute, operated by the Association of Universities for Research in Astronomy, Inc., from NASA contract NAs5-26555, and is reproduced with permission from AURA/STScI. Digital renditions of images produced by AURA/STScI are obtainable royalty-free. Credits: J.P. Harrington and K.J. Orkowski (University of Maryland), and NASA.)

This table summarizes display methods for the three types of images.

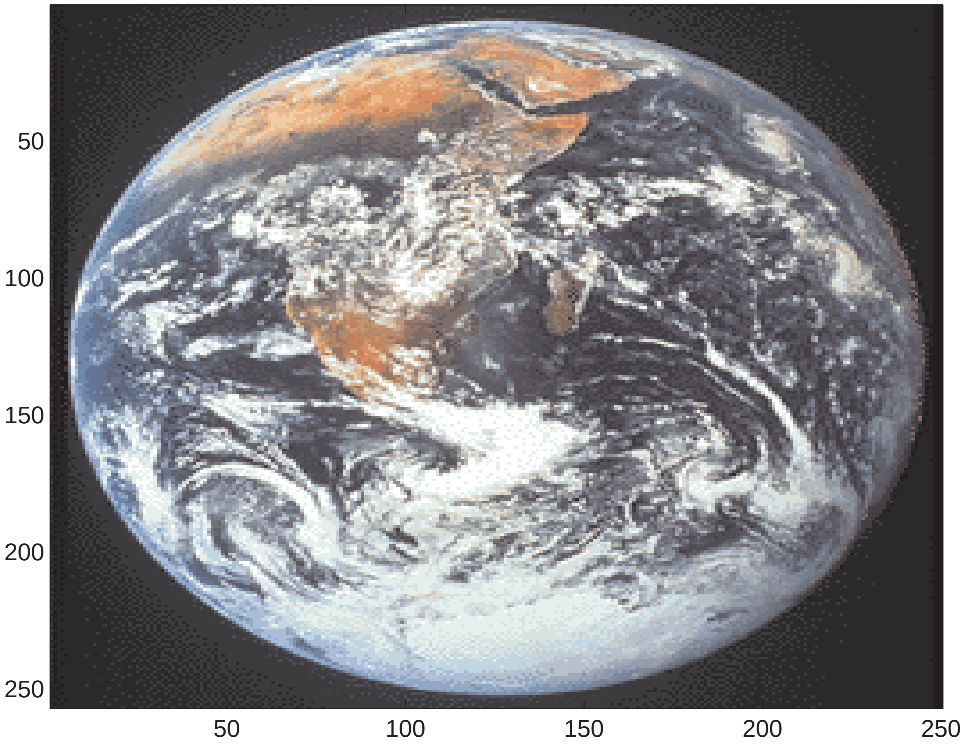
Image Type	Display Commands	Uses Colormap Colors
Indexed	`image(X);` `colormap(map)`	Yes
Intensity	`imagesc(I,[0 1]);` `colormap(gray)`	Yes
RGB (truecolor)	`image(RGB)`	No

Controlling Aspect Ratio and Display Size

The `image` function displays the image in a default-sized figure and axes. The image stretches or shrinks to fit the display area. Sometimes you want the aspect ratio of the display to match the aspect ratio of the image data matrix. The easiest way to do this is with the `axis image` command.

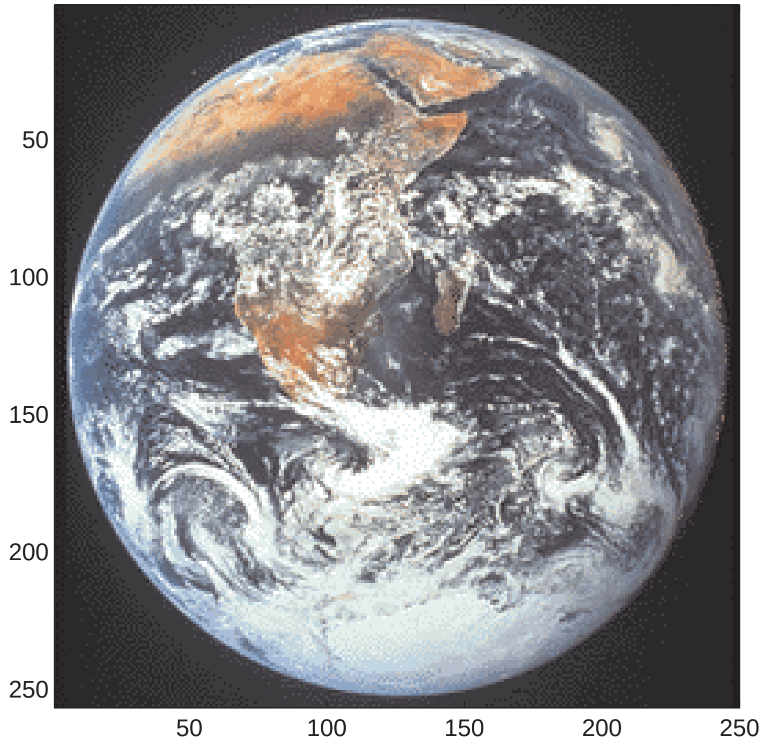
For example, these commands display the `earth` image in the `demos` directory using the default figure and axes positions:

```
load earth
image(X); colormap(map)
```



The elongated globe results from stretching the image display to fit the axes position. Use the `axis image` command to force the aspect ratio to be one-to-one.

```
axis image
```

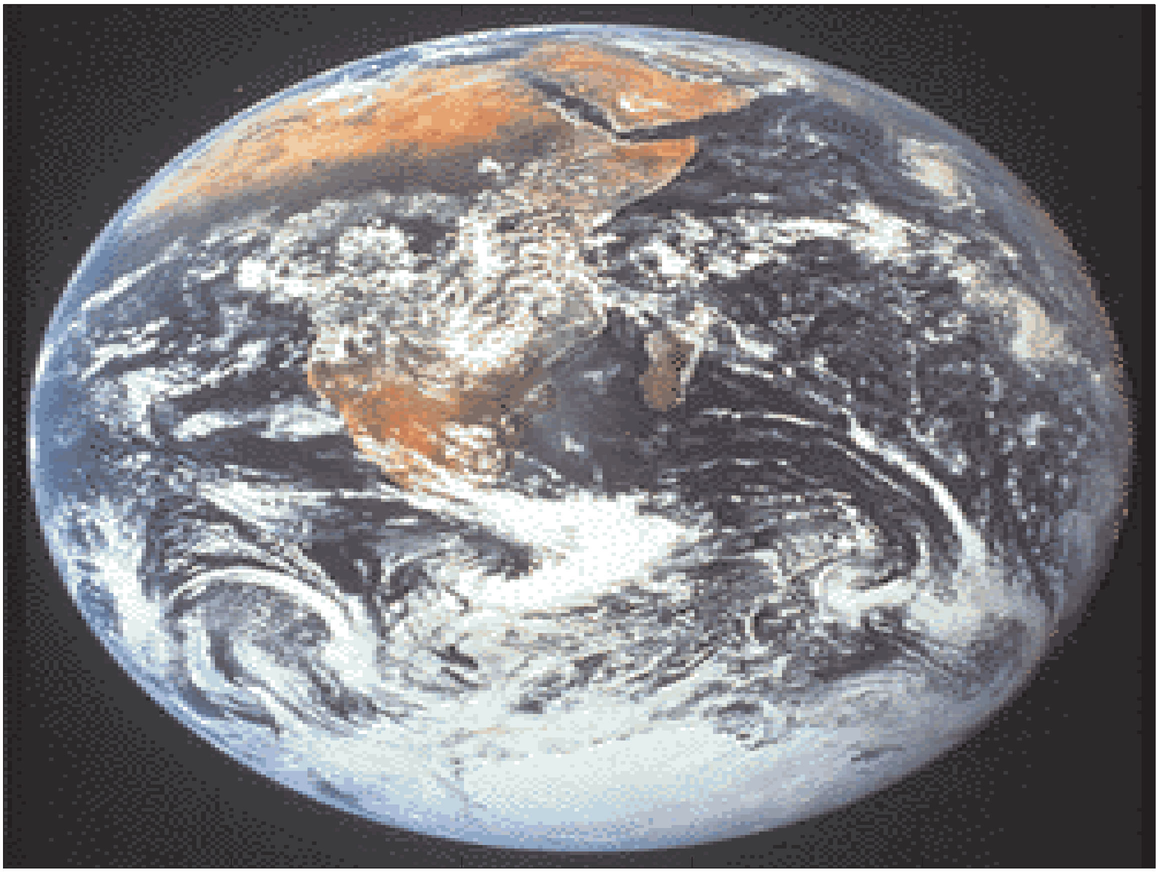


The axis image command works by setting the DataAspectRatio property of the axes object to [1 1 1]. See axis and axes for more information on how to control the appearance of axes objects.

Sometimes you want to display an image so that each element in the data matrix corresponds to a single screen pixel. To display an image with this one-to-one matrix-element-to-screen-pixel mapping, you need to resize the figure and axes. For example, these commands display the earth image so that one data element corresponds to one screen pixel:

```
[m,n] = size(X);
figure('Units','pixels','Position',[100 100 n m])
image(X); colormap(map)
set(gca,'Position',[0 0 1 1])
```

The figure's `Position` property is a four-element vector that specifies the figure's location on the screen as well as its size. The `figure` command positions the figure so that its lower left corner is at position (100,100) on the screen and so that its width and height match the image width and height. Setting the axes position to [0 0 1 1] in normalized units creates an axes that fills the figure. The resulting picture is shown.



The Image Object and Its Properties

In this section...

Image CData

Note The `image` and `imagesc` commands create image objects. Image objects are children of axes objects, as are line, patch, surface, and text objects. Like all Handle Graphics objects, the image object has a number of properties you can set to fine-tune its appearance on the screen. The most important properties of the image object with respect to appearance are `CData`, `CDataMapping`, `XData`, and `YData`. These properties are discussed in this and the following sections. For detailed information about these and all the properties of the image object, see `image`.

The `CData` property of an image object contains the data array. In the following commands, `h` is the handle of the image object created by `image`, and the matrices `X` and `Y` are the same:

```
h = image(X); colormap(map)
Y = get(h,'CData');
```

The dimensionality of the `CData` array controls whether the image displays using colormap colors or as an RGB image. If the `CData` array is two-dimensional, the image is either an indexed image or an intensity image; in either case, the image is displayed using colormap colors. If, on the other hand, the `CData` array is *m*-by-*n*-by-3, it displays as a truecolor image, ignoring the colormap colors.

Image CDataMapping

The `CDataMapping` property controls whether an image is `indexed` or `intensity`. To display an indexed image set the `CDataMapping` property to `'direct'`, so that the values of the `CData` array are used directly as indices into the figure's colormap. When the `image` command is used with a single input argument, it sets the value of `CDataMapping` to `'direct'`:

```
h = image(X); colormap(map)
get(h,'CDataMapping')
ans =

direct
```

Intensity images are displayed by setting the `CDataMapping` property to `'scaled'`. In this case, the `CData` values are linearly scaled to form colormap indices. The axes `CLim` property controls the scale factors. The `imagesc` function creates an image object whose `CDataMapping` property is set to `'scaled'`, and it adjusts the `CLim` property of the parent axes. For example:

```
h = imagesc(I,[0 1]); colormap(map)
get(h,'CDataMapping')
ans =

scaled

get(gca,'CLim')
ans =

[0 1]
```

XData and YData

The `XData` and `YData` properties control the coordinate system of the image. For an m-by-n image, the default `XData` is `[1 n]` and the default `YData` is `[1 m]`. These settings imply the following:

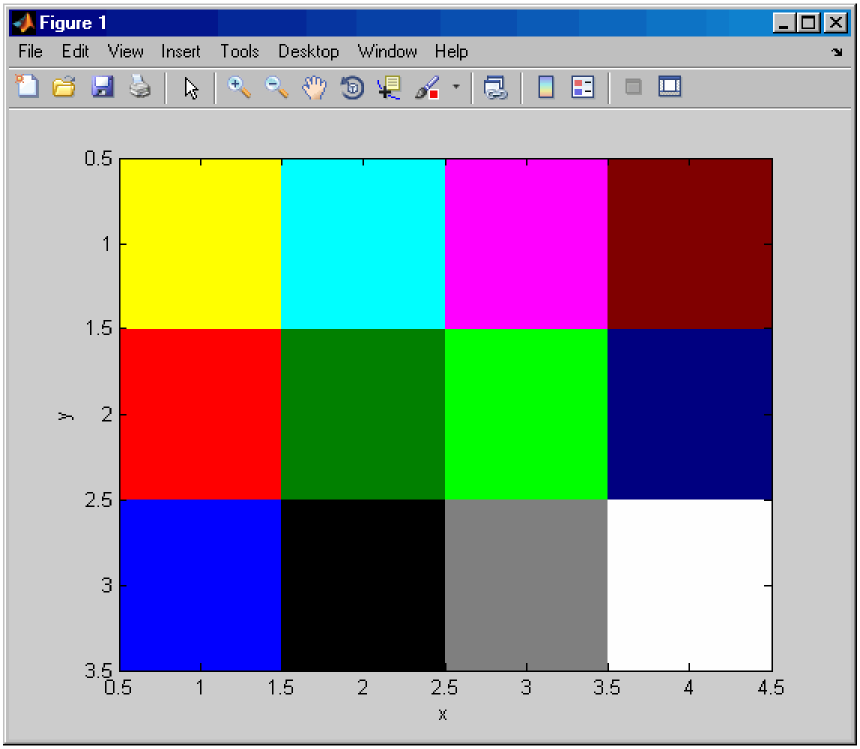
- The left column of the image has an x-coordinate of 1.
- The right column of the image has an x-coordinate of n.
- The top row of the image has a y-coordinate of 1.

- The bottom row of the image has a *y*-coordinate of *m*.

For example, the statements

```
X = [1 2 3 4; 5 6 7 8; 9 10 11 12];
h = image(X); colormap(colorcube(12))
xlabel x; ylabel y
```

produce the following picture.



The XData and YData properties of the resulting image object have the following default values:

```
get(h,'XData')
ans =
```

```
        1     4

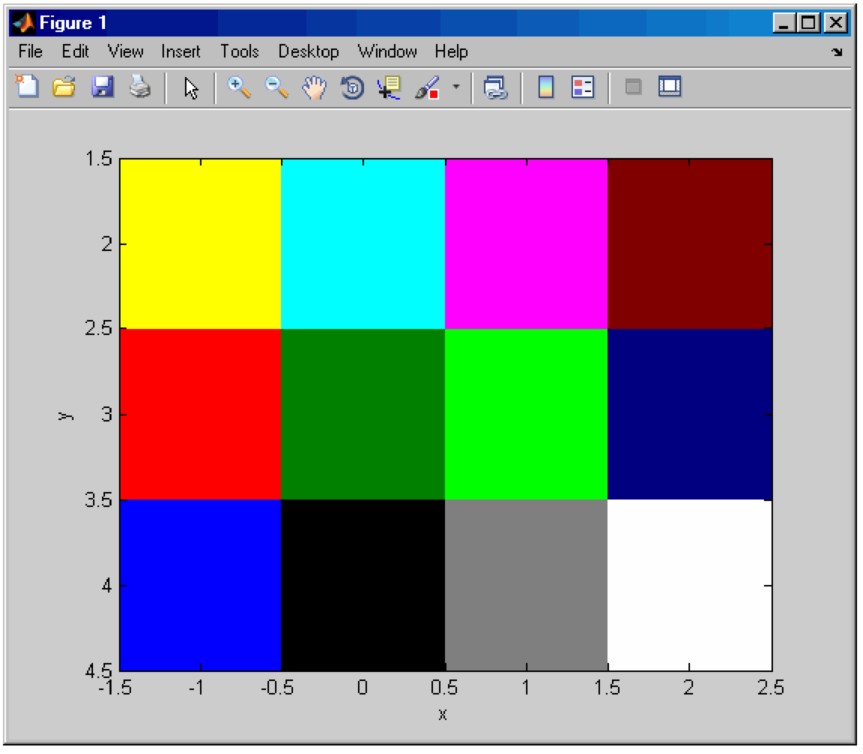
get(h,'YData')
ans =

        1     3
```

However, you can override the default settings to specify your own coordinate system. For example, the statements

```
X = [1 2 3 4; 5 6 7 8; 9 10 11 12];
image(X,'XData',[-1 2],'YData',[2 4]); colormap(colorcube(12))
xlabel x; ylabel y
```

produce the following picture.



Adding Text to Images

Use basic array indexing to rasterize text strings into an existing image, as described in this section:

Draw the text strings using `text`, and then capture a bitmapped version of them using `getframe`. Then find the black pixels and convert their subscripts to indexes using `sub2ind`. Use these subscripts to “paint” the text into the image into which you want to add the text string, and then save that image. Here is an example using the image in the demo MAT-file `mandrill.mat`:

```
% Create the text in an axis:
t = text(.05,.1,'Mandrill Face', ...
         'FontSize',12, 'FontWeight','demi');

% Capture the text from the screen:
F = getframe(gca,[10 10 200 200]);

% Close the figure:
close

% Select any plane of the resulting RGB image:
c = F.cdata(:,:,1);

% Note: If you have Image Processing Toolbox installed,
% you can convert the RGB data from the frame to black or white:
% c = rgb2ind(F.cdata,2);

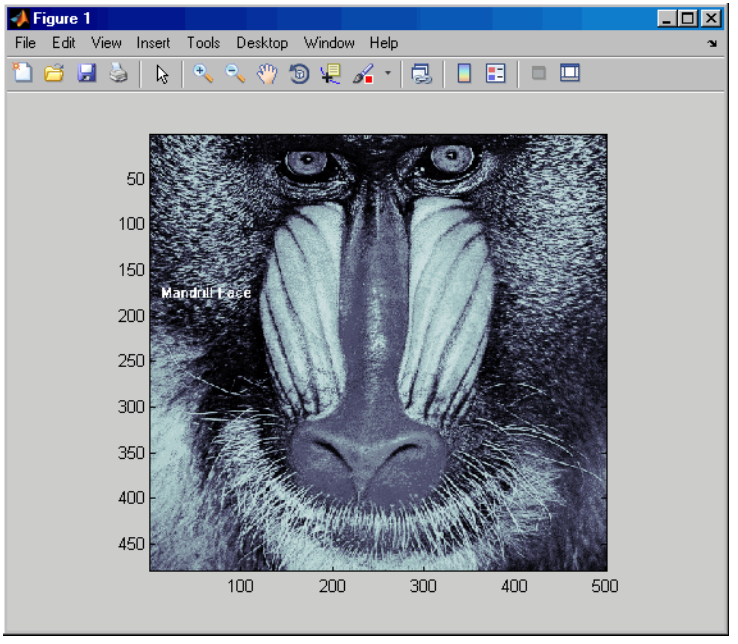
% Determine where the text was (black is 0):
[i,j] = find(c == 0);

% Read in or load the image that is to contain the text:
load mandrill

% Use the size of that image, plus the row/column locations
% of the text, to determine locations in the new image:
ind = sub2ind(size(X),i,j);

% Index into new image, replacing pixels with white:
X(ind) = uint8(255);
```

```
% Display and color the new image:
imagesc(X)
axis image
colormap(bone)
```



Additional Techniques for Fast Image Updating

To increase the rate at which the CData property of an image object updates, optimize CData and set some related figure and axes properties:

- Use the smallest data type possible. Using a uint8 data type for your image will be faster than using a double data type.

  Part of the process of setting the image's CData property includes copying the matrix for the image's use. The overall size of the matrix is dependent on the size of its individual elements. Using smaller individual elements (i.e., a smaller data type) decreases matrix size, and reduces the amount of time needed to copy the matrix.

- Use the smallest acceptable matrix.

If the speed at which the image is displayed is your highest priority, you may need to compromise on the size and quality of the image. Again, decreasing the size reduces the time needed to copy the matrix.

- Make the axes exactly the same size (in pixels) as the `CData` matrix.

  Maintaining a one-to-one correspondence between the data and the onscreen pixels eliminates the need for interpolation. For example:

```
set(gca,'Units','pixels')
pos = get(gca,'Position')
width = pos(3);
height = pos(4);
```

  When the size of your `CData` exactly equals `[width height]`, each element of the array corresponds directly to a pixel. Otherwise, the values in the `CData` array must be interpolated so the image fits the axes at their current size.

- Set the limit mode properties (`XLimMode` and `YLimMode`) of your axes to `manual`.

  If they are set to `auto`, then every time an object (such as an image, line, patch, etc.) changes some aspect of its data, the axes must recalculate its related properties. For example, if you specify

```
image(firstimage);
set(gca, 'xlimmode','manual',...
'ylimmode','manual',...
'zlimmode','manual',...
'climmode','manual',...
'alimmode','manual');
```

  the axes do not recalculate any of the limit values before redrawing the image.

- Set the figure's `DoubleBuffer` property to `off`.

```
set(gcf,'doublebuffer','off');
```

  The `DoubleBuffer` property is `'on'` by default, producing flicker-free animation. To maximize rendering speed, set `DoubleBuffer` to `'off'`.

- Alternately, consider using a `movie` object if the main point of your task is to simply display a series of images onscreen.

  The MATLAB `movie` object utilizes underlying system graphics resources directly, instead of executing MATLAB object code. This is faster than repeatedly setting an image's `CData` property, as described earlier.

Printing Images

When you set the axes `Position` to `[0 0 1 1]` so that it fills the entire figure, the aspect ratio is not preserved when you print because MATLAB printing software adjusts the figure size when printing according to the figure's `PaperPosition` property. To preserve the image aspect ratio when printing, set the figure's `PaperPositionMode` to `'auto'` from the command line.

```
set(gcf,'PaperPositionMode','auto')
print
```

When `PaperPositionMode` is set to `'auto'`, the width and height of the printed figure are determined by the figure's dimensions on the screen, and the figure position is adjusted to center the figure on the page. If you want the default value of `PaperPositionMode` to be `'auto'`, enter this line in your `startup.m` file.

```
set(0,'DefaultFigurePaperPositionMode','auto')
```

Printed images may not always be the same size as they are on your monitor. The size depends on accurately specifying the numbers of pixels per inch that you monitor is displaying.

To specify the pixels-per-inch on your display, do the following (in Microsoft Windows):

1. Go into your Display Properties by right-clicking on an empty space on your desktop and choose **Properties**.

2. Click the **Settings** pane.

3. Click the **Advanced** button and choose the General pane.

4. Switch DPI setting to Custom setting and hold a real ruler up to the picture of the ruler on the screen and drag until they match.

Until you do this, neither Windows software nor any other can determine how big images on the screen are, and printed images cannot match the size.

On the Macintosh® platform, pixels per inch is hard-coded to 72.

Converting the Data or Graphic Type of Images

Converting between data types changes the interpretation of the image data. If you want the resulting array to be interpreted properly as image data, rescale or offset the data when you convert it. (See the earlier sections "Image Types" on page 6-5 and "8-Bit and 16-Bit Indexed Images" on page 6-10 for more information about offsets.)

For certain operations, it is helpful to convert an image to a different image type. For example, to filter a color image that is stored as an indexed image, first convert it to RGB format. To do this efficiently, use the `ind2rgb` function. When you apply the filter to the RGB image, the intensity values in the image are filtered, as is appropriate. If you attempt to filter the indexed image, the filter is applied to the indices in the indexed image matrix, and the results may not be meaningful.

You can also perform certain conversions just using MATLAB syntax. For example, to convert a grayscale image to RGB, concatenate three copies of the original matrix along the third dimension:

```
RGB = cat(3,I,I,I);
```

The resulting RGB image has identical matrices for the red, green, and blue planes, so the image appears as shades of gray.

Changing the graphics format of an image, perhaps for compatibility with another software product, is very straightforward. For example, to convert an image from a BMP to a PNG, load the BMP using `imread`, set the data type to `uint8`, `uint16`, or `double`, and then save the image using `imwrite`, with `'PNG'` specified as your target format. See `imread` and `imwrite` for the specifics of which bit depths are supported for the different graphics formats, and for how to specify the format type when writing an image to file.

7

Printing and Exporting

Overview of Printing and Exporting

Print and Export Operations

There are four basic operations that you can perform in printing or transferring figures you've created with MATLAB graphics to specific file formats for other applications to use.

Operation	Description
Print	Send a figure from the screen directly to the printer.
Print to File	Write a figure to a PostScript® file to be printed later.
Export to File	Export a figure in graphics format to a file, so that you can import it into an application.
Export to Clipboard	Copy a figure to the Microsoft Windows clipboard, so that you can paste it into an application.

Graphical User Interfaces

In addition to typing MATLAB commands, you can use interactive tools for either Microsoft Windows or UNIX® to print and export graphics. The table below lists the GUIs available for doing this and explains how to open them from figure windows.

Dialog Box	How to Open	Description
Print (Windows and UNIX)	**File > Print** or `printdlg` function	Send figure to the printer, select the printer, print to file, and several other options

Dialog Box	How to Open	Description
Print Preview	**File > Print Preview** or `printpreview` function	View and adjust the final output
Export	**File > Export**	Export the figure in graphics format to a file
Copy Options	**Edit > Copy Options**	Set format, figure size, and background color for Copy to Clipboard
Figure Copy Template	**File > Preferences**	Change text, line, axes, and UI control properties

You can open the Print and Print Preview dialog boxes from an M-file or from the command line with the `printdlg` and `printpreview` functions.

Command Line Interface

You can print a MATLAB figure from the command line or from an M-file. Use the `set` function to set the properties that control how the printed figure looks. Use the `print` function to specify the output format and start the print or export operation.

Note Printed output from MATLAB commands and Print Previews of it are not guaranteed to duplicate the look of figures on your display screen in every detail. Many factors, including the complexity of the figure, available fonts, and whether a native printer driver or a MATLAB built-in driver to is used, affect the final output and can cause printed output to differ from what you see on your screen.

Modifying Properties with set

The `set` function changes the values of properties that control the look of a figure and objects within it. These properties are stored with the figure; some are also properties of children such as axes or annotations. When you change one of the properties, the new value is saved with the figure and affects the look of the figure each time you print it until you change the setting again.

To change the print properties of the current figure, the `set` command has the form

```
set(gcf, 'Property1', value1, 'Property2', value2, ...)
```

where `gcf` is a function call that returns the handle of the current figure, and each property value pair consists of a named property followed by the value to which the property is set.

For example,

```
set(gcf, 'PaperUnits', 'centimeters', 'PaperType', 'A4', ...)
```

sets the units of measure and the paper size. "Changing a Figure's Settings" on page 7-43 describes commonly used print properties. The Figure Properties reference page contains a complete list of the properties.

Examining Properties with get

You can also use the `get` function to retrieve the value of a specific property.

```
a = get(gcf, 'Property')
```

Note You can also peruse and modify figure and other object properties with the Property Inspector GUI, which you can open with the `inspect` command. To open the current figure in the Property Inspector, type
`inspect(gcf)`

Printing and Exporting with print

The `print` function performs any of the four actions shown in the table below. You control what action is taken, depending on the presence or absence of certain arguments.

Action	Print Command
Print a figure to a printer	`print`
Print a figure to a file for later printing	`print` *`filename`*

Action	Print Command
Copy a figure in graphics format to the clipboard	`print -dfileformat`
Export a figure to a graphics format file that you can later import into an application	`print -dfileformat filename`

You can also include optional arguments with the `print` command. For example, to export Figure No. 2 to file `spline2d.eps`, with 600 dpi resolution, and using the EPS color graphics format, use

```
print -f2 -r600 -depsc spline2d
```

The functional form of this command is

```
print('-f2', '-r600', '-depsc', 'spline2d');
```

Printing on UNIX Platforms without a Display

If you run with the PostScript `-nodisplay` startup option, or run without the `DISPLAY` environment variable set, you can use most `print` options that apply to the UNIX platform, but some restrictions apply. For example, in `nodisplay` mode uicontrols do not print; thus you cannot print a GUI if you run in this mode.

See "Printing and Exporting without a Display" in the documentation for the `print` function for details.

Specifying Parameters and Options

The table below lists parameters you can modify for the figure to be printed or exported. To change one of these parameters, use the Print Preview or the UNIX Print dialog box, or use the `set` or `print` function.

See "Changing a Figure's Settings" on page 7-43 for more detailed instructions.

Parameter	Description
Figure size	Set size of the figure on printed page
Figure position	Set position of figure on printed page
Paper size	Select printer paper, specified by dimension or type
Paper orientation	Specify way figure is oriented on page
Position mode	Specify figure position yourself or let it be determined automatically
Graphics format	Select format for exported data (e.g., EPS, JPEG)
Resolution	Specify how finely your figure is to be sampled
Renderer	Select method (algorithm) for drawing graphics
Renderer mode	Specify the renderer yourself or automatically determine which renderer to use based on the figure's contents
Axes tick marks	Keep axes tick marks and limits as shown or automatically adjust them depending on figure size
Background color	Keep background color as shown on screen or force it to white
Line and text color	Keep line and text objects as shown on screen or print them in black and white
UI controls	Show or hide all user interface controls in figure
Bounding box	Leave space between outermost objects in plot and edges of its background area
CMYK	Automatically convert RGB values to CMYK values
Character set encoding	Select character set for PostScript printers

Default Settings and How to Change Them

If you have not changed the default print and export settings, MATLAB prints or exports the figure as follows:

- 8-by-6 inches with no window frame
- Centered, in portrait format, on 8.5-by-11 inch paper if available
- Using white background color for the figure and axes
- Scaling ticks and limits of the axes to accommodate the printed size

Setting Defaults for a Figure

In general, to change the property settings for a specific figure, follow the instructions given in the section "Changing a Figure's Settings" on page 7-43.

Any settings you change with the Print Preview and Print dialog boxes or with the `set` function are saved with the figure and affect each printing of the figure until you change the settings again.

The settings you change with the **Figure Copy Template Preferences** and **Copy Options Preferences** panels alter the figure as it is displayed on the screen.

Setting Defaults for the Session

You can set the session defaults for figure properties. Set the session default for a property using the syntax

```
set(0, 'DefaultFigurepropertyname', 'value')
```

where *`propertyname`* is one of the named figure properties. This example sets the paper orientation for all subsequent print operations in the current MATLAB session.

```
set(0, 'DefaultFigurePaperOrientation', 'landscape')
```

The Figure Properties reference page contains a complete list of the properties.

To see what default properties you can set that will be applied to all subsequent figures in the same MATLAB session, type

```
set(0,'default')
```

To see their current settings, type

```
get(0,'default')
```

Setting Defaults Across Sessions

You can set the session-to-session defaults for figure properties, the print driver, and the `print` function.

Print Device and Print Command. Set the default print driver and the default print command in your `printopt.m` file. This file contains instructions for changing these settings and for displaying the current defaults. Open `printopt.m` in your editor by typing the command

```
edit printopt
```

Scroll down about 40 lines until you come to this comment line:

```
%---> Put your own changes to the defaults here (if needed)
```

Add your changes after that line. For example, to change the default driver, first find the line that sets `dev`, and then replace the text string with an appropriate value. So, to set the default driver to HP LaserJet III, modify the line to read

```
dev = '-dljet3';
```

For the full list of values for `dev`, see the Drivers section of the `print` reference page.

Note If you set `dev` to be a graphics format, such as `-djpeg`, the figure is exported to that type of file rather than being printing.

Figure Properties. Set the session-to-session default for a property by including commands like the following in your `startup.m` file:

```
set(0, 'DefaultFigurepropertyname', 'value')
```

where *`propertyname`* is one of the named figure properties. For example,

```
set(0, 'DefaultFigureInvertHardcopy', 'off')
```

keeps the figure background in the screen color.

This is the same command you use to change a session default, except by adding it to your `startup.m` file, it executes automatically every time you launch MATLAB.

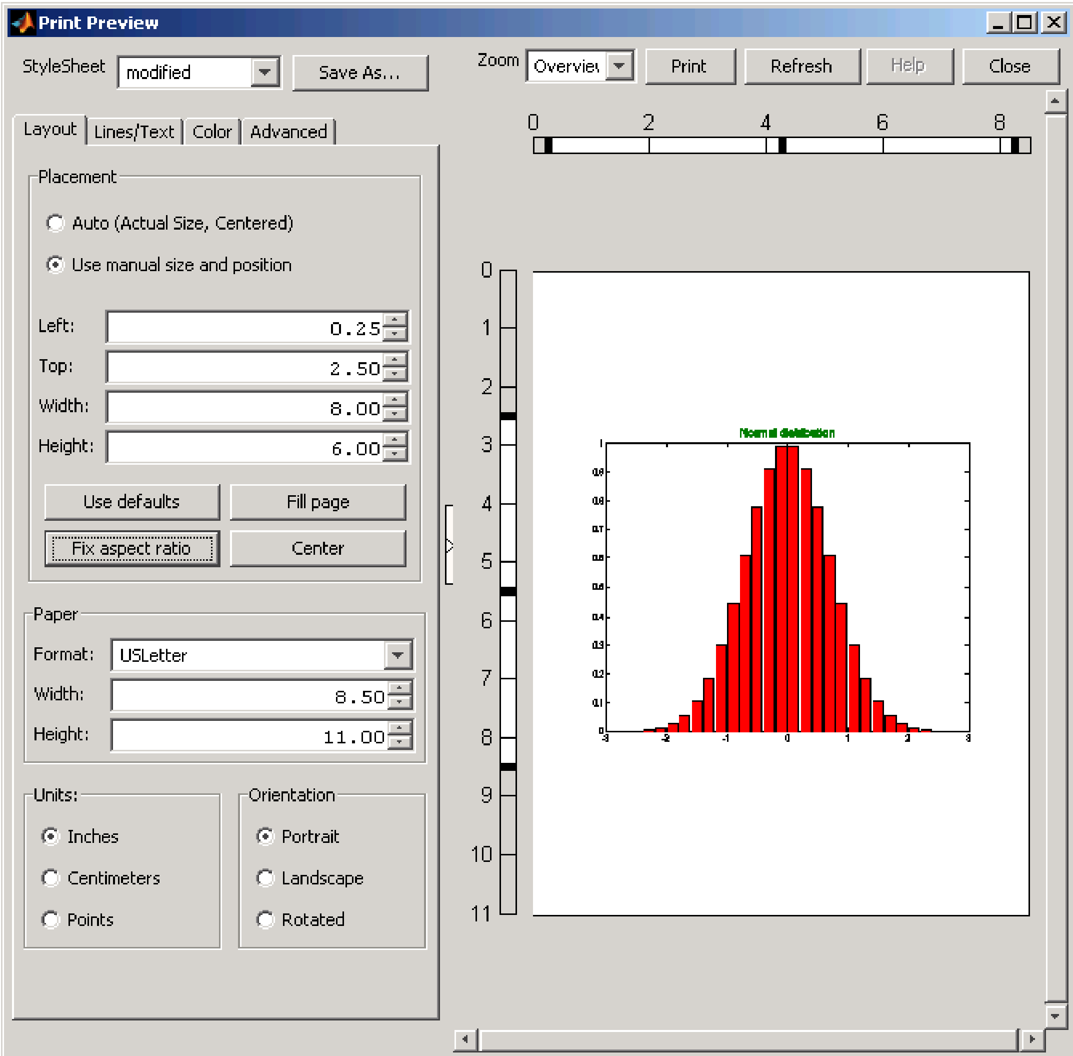
Note Options you specify in arguments to the `print` command override properties set using MATLAB commands or the Print Preview dialog box, which in turn override any MATLAB default settings specified in `printopt.m` or `startup.m`.

How to Print or Export

In this section...

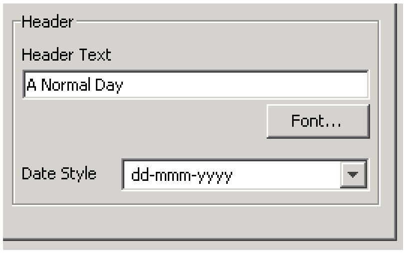
Using Print Preview

Before you print or export a figure, preview the image by selecting **Print Preview** from the figure window’s **File** menu. If necessary, you can use the `set` function to adjust specific characteristics of the printed or exported figure. Adjustments that you make in the Print Preview dialog also set figure properties; these changes can affect the output you get should you print the figure later with the `print` command. See “Changing a Figure’s Settings” on page 7-43 for details.

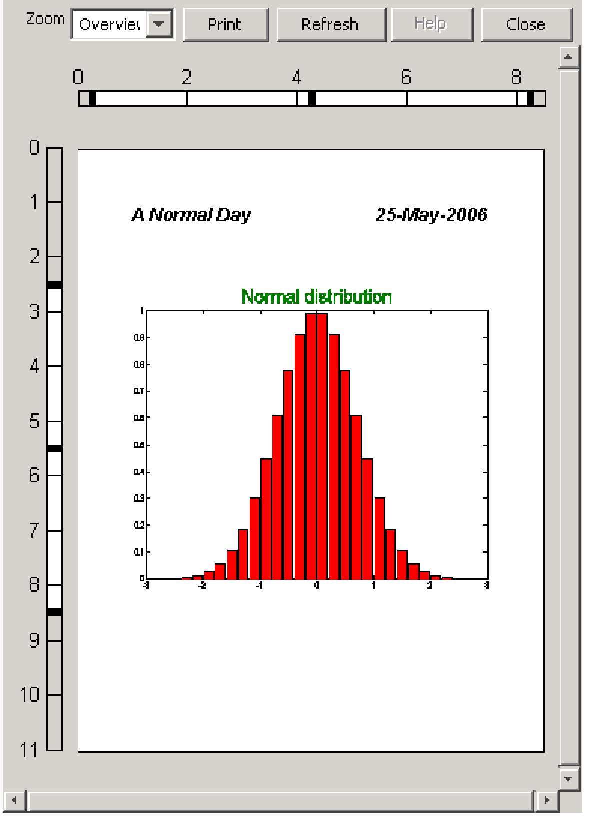


Adding a Header to the Printed Page

You can add a header to the page you are about to print by clicking the **Lines/Text** tab at the top of the Print Preview dialog box. At the bottom of that panel are the **Header** controls, as shown here:



The print header includes any text you want to appear at the top of the printed page. It can also include the current date. In the **Header Text** edit box, enter the text of the header. Under `Date Type`, select from a number of possible formats with which to display the current date and/or time. The default is to include no date. Click the **Font** button to change the font, font style, font size, or script type for the header text and date format. If you don't see the header as you specified it, click the **Refresh** button over the preview pane. A page containing a header plus date in bold italics is shown in the preview below:



Click **Print** to open the standard print dialog box to print the page. Click **Close** to close the dialog box and apply these settings to your figure.

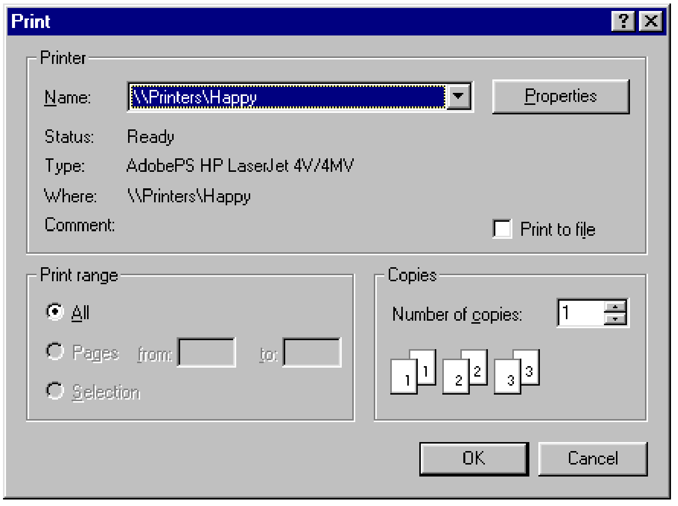
Printing a Figure

This section tells you how to print your figure to a printer:

- "Printing with the Print GUI on Microsoft Windows" on page 7-14
- "Printing with the Print GUI on UNIX Platforms" on page 7-15
- "Printing Using PostScript Commands" on page 7-18

Printing with the Print GUI on Microsoft Windows

MATLAB printing on Windows platforms uses the standard Windows Print dialog box, which most Windows software products share. To open the Windows Print dialog box, select **Print** from the figure window's **File** menu or click the **Print** button in the Print Preview dialog box.



- To print a figure, first select a printer from the list box, then click **OK**.
- To save it to a file, click the **Print to file** check box, click **OK**, and when the Print to File window appears, enter the filename you want to save the figure to. The file is written to your current working directory.

Settings you can change in the Windows Print dialog box are as follows:

Properties. To make changes to settings specific to a printer, click the **Properties** button. This opens the Windows Document Properties window.

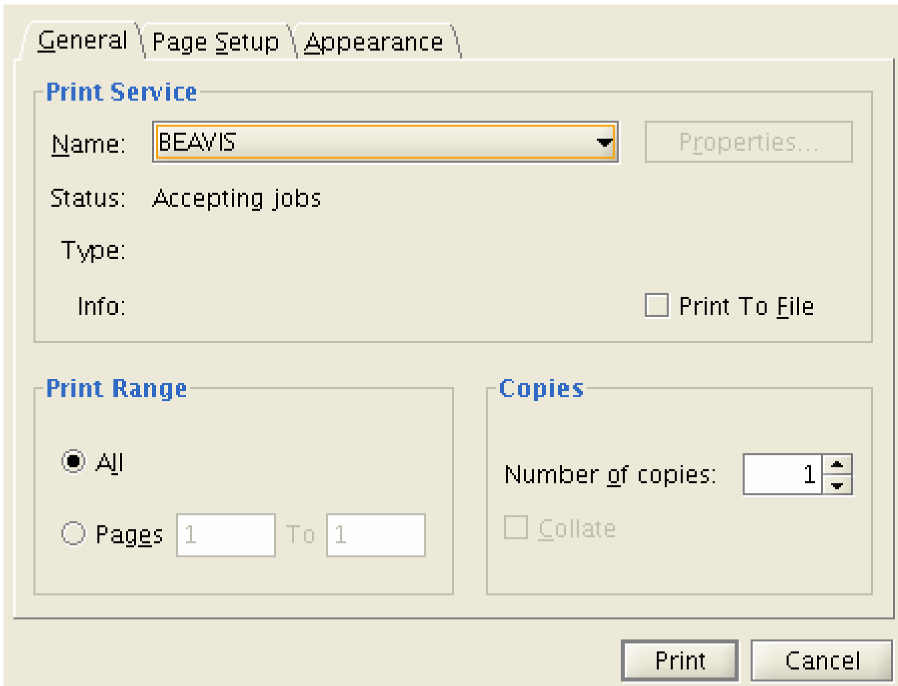
Print range. You can only select **All** in this panel. The selection does not affect your printed output.

Copies. Enter the number of copies you want to print.

You can also open the Print dialog programmatically via the `printdlg` function.

Printing with the Print GUI on UNIX Platforms

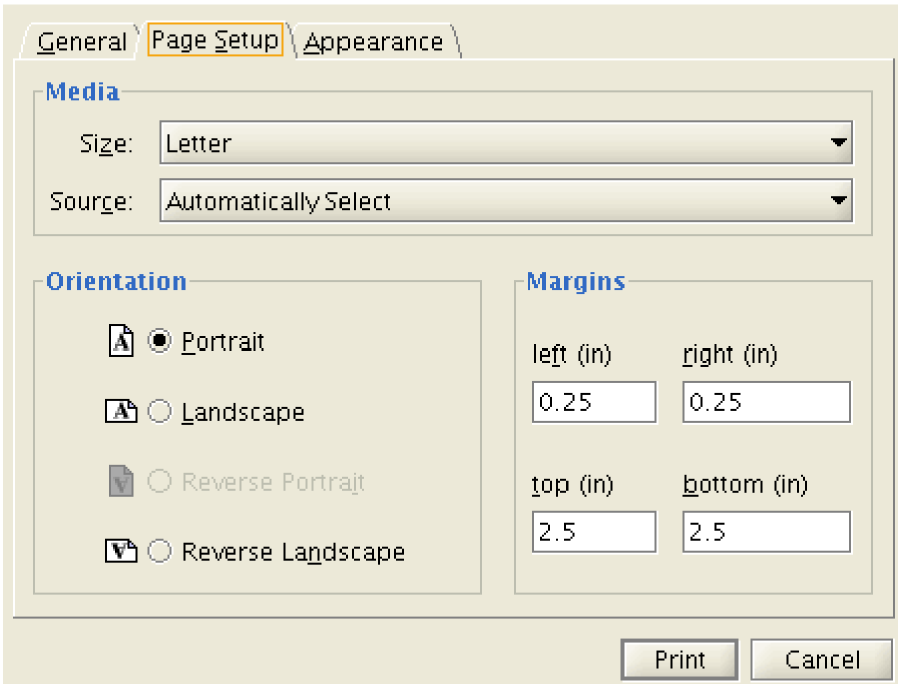
MATLAB printing on UNIX platrforms has a Print dialog box containing three tabs. To open the Print dialog box, select **Print** from the figure window's **File** menu. It opens showing the **General** tab's contents:



To print a figure, click the **Name** button under **Print Service** and select a printer from the list box.

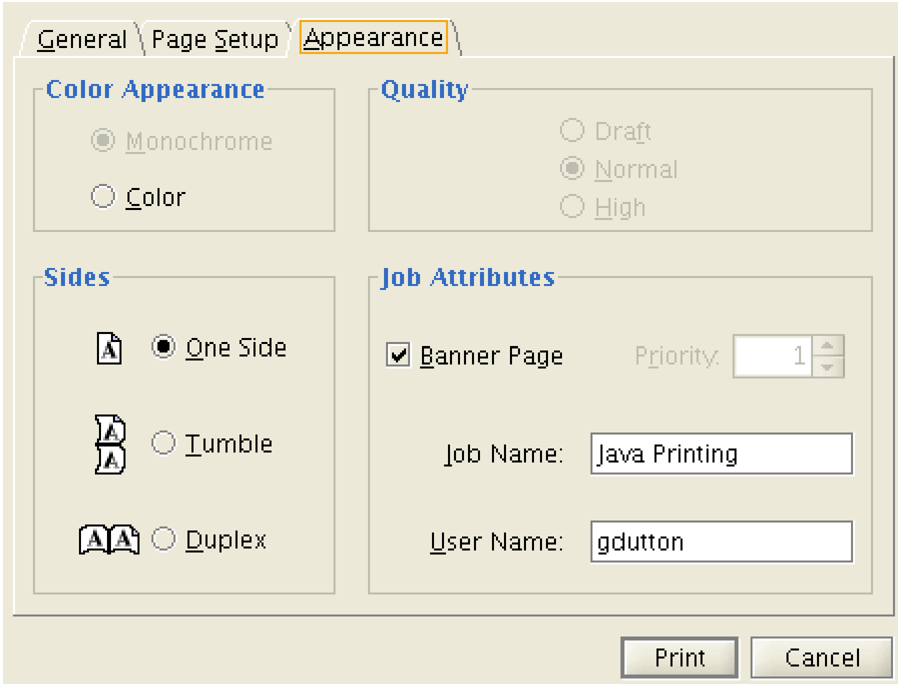
Note Printers accessed from the Print dialog are assumed to be PostScript-enabled. If you want to print to a non-PostScript device, you will need to use **File > Save As** and specify the **Save as type** or issue a `print` command specifying the appropriate driver with the `-d` flag.

The **Page Setup** tab on the Print dialog looks like this:



You can set paper characteristics and margins with the controls on this tab. You might want to use the Print Preview dialog instead, however, as it allows you to do the same things and gives you visual feedback at the same time. For details, see "Using Print Preview" on page 7-10.

The **Appearance** Print dialog tab lets you control several aspects of your print jobs:



The **Appearance** options include Duplex and Tumble printing, whether a banner page should precede the printed page, whether to print in color, and what quality of printing to use. You can also use **Print Preview** to control color.

Related settings in the Print Preview dialog box include

Printing in Color. Depending on the capabilities of the printer you are using, you can print in black and white, grayscale, or color by selecting the appropriate button in the **Color Scale** panel of the Print Preview **Color** tab. You can also choose a background color that is the same or different from the figure's color.

Figure Size and Position on Printed Page. If you want the printed plot to have the same size as it does on your screen, select **Auto (Actual Size, Centered)** on the **Layout** tab. If you want the printed output to have a specific size, select **Use manual size and position**.

See "Setting the Figure Size and Position" on page 7-47 for more information.

Axes Limits and Ticks. To force the same number of ticks and the same limit values for the axes as are used on the screen to be printed, select **Keep screen limits and ticks** on the **Advanced** tab of the Print Preview dialog box. To automatically scale the limits and ticks of the axes based on the size of the printed figure, select **Recompute limits and ticks**.

See "Setting the Axes Ticks and Limits" on page 7-60 for more information.

Printing Using PostScript Commands

Use the `print` function to print from the PostScript command line or from a program. See "Printing and Exporting with print" on page 7-4 for more information.

To send the current or most recently active figure to a printer, simply type

```
print
```

The Printing Options table on the `print` reference page shows a full list of options that you can use with the `print` function. For example, the following command prints Figure No. 2 with 600 dpi resolution, using the Canon BubbleJet BJ200 printer driver:

```
print -f2 -r600 -dbj200
```

Printing to a File

Instead of sending your figure to the printer right now, you have the option of "printing" it to a file, and then sending the file to the printer later on. You can also append additional figures to the same file using the `print` command.

Note When you print to a file, the file name must have fewer than 128 characters, including path name. When you print to a file in your current directory, the filename must have fewer than 126 characters, because MATLAB places `'./'` or `'.\'`' at the beginning of the filename when referring to it.

This section tells you how to save your figure to a file:

- "Printing to a File with the Print GUI on Windows Platforms" on page 7-19
- "Printing to a File with the Print GUI on UNIX Platforms" on page 7-19
- "Printing to a File Using MATLAB Commands" on page 7-19

Printing to a File with the Print GUI on Windows Platforms

1. To open the Print dialog box, select **Print** from the figure window's **File** menu.
2. Select the check box labeled **Print to file**, and click the **OK** button.
3. The **Print to file** dialog box appears, allowing you to specify the output directory and filename.

Printing to a File with the Print GUI on UNIX Platforms

1. To open the Print dialog box, select **Print** from the figure window's **File** menu.
2. Select the radio button labeled **File**, and either fill in or browse for the directory and filename.

Printing to a File Using MATLAB Commands

To print the figure to a PostScript file, type

```
print filename
```

If you don't specify the filename extension, MATLAB uses an extension that is appropriate for the print driver being used.

You can also include an `-options` argument when printing to a file. For example, to append the current figure to an existing file, type

```
print -append filename
```

The only way to append to a file is by using the `print` function. There is no dialog box that enables you to do this.

Note If you print a figure to a file, the file can only be printed and cannot be imported into another application. If you want to create a figure file that you can import into an application, see the next section, "Exporting to a File"

Appending Additional Figures to a File. Once you have printed one figure to a PostScript file, you can append other figures to that same file using the `-append` option of the `print` function. You can only append using the `print` function.

This example prints Figure No. 2 to PostScript file `myfile.ps`, and then appends Figure No. 3 to the end of the same file:

```
print -f2 myfile
print -f3 -append myfile
```

Exporting to a File

Export a figure in a graphics format to a file if you want to import it into another application, such as a word processor. You can export to a file from the Windows or UNIX Export Setup dialog box or from the command line.

This section tells you how to export your figure to a file:

- "Using the Export Setup GUI" on page 7-21
- "Exporting Using MATLAB Commands" on page 7-28

It also covers

- "Exporting with getframe" on page 7-29
- "Saving Multiple Figures to an AVI File" on page 7-30
- "Importing MATLAB Graphics into Other Applications" on page 7-30

For further information, see "Choosing a Graphics Format" on page 7-73.

Note When you export to a file, the file name must have fewer than 128 characters, including path name. When you print to a file in your current directory, the filename must have fewer than 126 characters, because MATLAB places `'./'` or `'.\'`' at the beginning of the filename when referring to it.

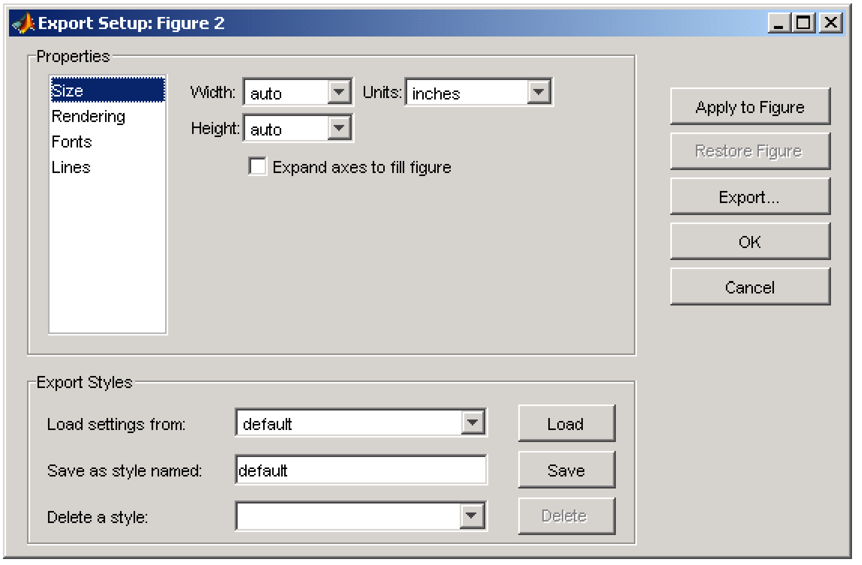
Using the Export Setup GUI

The Export Setup GUI appears when you select **Export Setup** from the **File** menu of a figure window. This GUI has four dialog boxes that enable you to adjust the size, rendering, font, and line appearance of your figure prior to exporting it. You select each of these dialog boxes by clicking **Size**, **Rendering**, **Fonts**, or **Lines** from the**Properties** list. For a description of each dialog box, see

- "Adjusting the Figure Size" on page 7-21
- "Changing the Rendering" on page 7-22
- "Changing Font Characteristics" on page 7-24
- "Changing Line Characteristics" on page 7-25

Adjusting the Figure Size

Click **Size** in the Export Setup dialog box to display this dialog box.

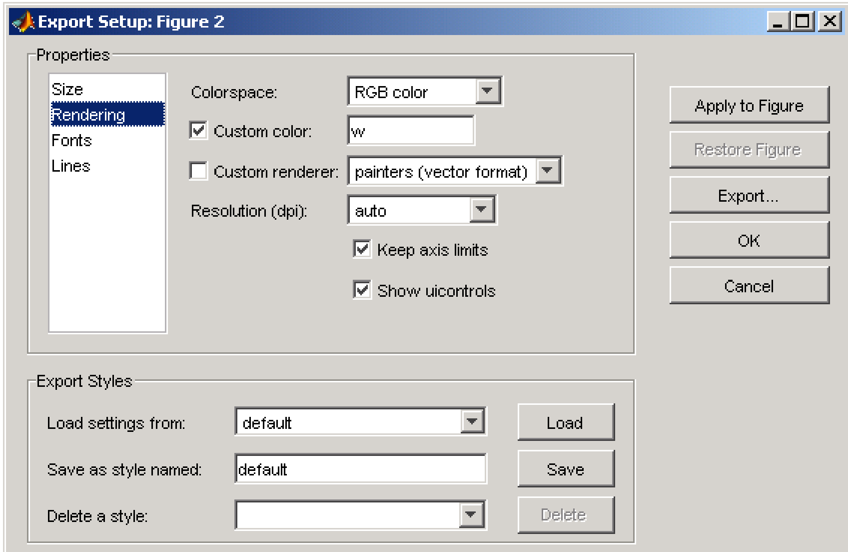


The Size dialog box modifies the size of the figure as it will appear when imported from the export file into your application. If you leave the **Width** and **Height** settings on `auto`, the figure remains the same size as it appears on your screen. You can change the size of the figure by entering new values in the **Width** and **Height** text boxes and then clicking **Apply to Figure**. To go back to the original settings, click **Restore Figure**.

To save any settings that you change, or to load settings that you used earlier, see "Saving and Loading Settings" on page 7-27.

Changing the Rendering

Click **Rendering** in the Export Setup dialog box to display this dialog box.



You can change the settings in this dialog box as follows:

Colorspace. Use the drop-down list to select a colorspace. Your choices are

- `Black and white`
- `Grayscale`
- `RGB color`
- `CMYK color`

Custom Color. Click the check box and enter a color to be used for the figure background. Valid entries are

- `white`, `yellow`, `magenta`, `red`, `cyan`, `green`, `blue`, or `black`
- Abbreviated name for the same colors — `w`, `y`, `m`, `r`, `c`, `g`, `b`, `k`
- Three-element RGB value — See the help for `colorspec` for valid values. Examples: `[1 0 1]` is magenta. `[0 .5 .4]` is a dark shade of green.

Custom Renderer. Click the check box and select a renderer from the drop-down list:

- painters (vector format)
- OpenGL (bitmap format)
- Z-buffer (bitmap format)

Resolution. You can select one of the following from the drop-down list:

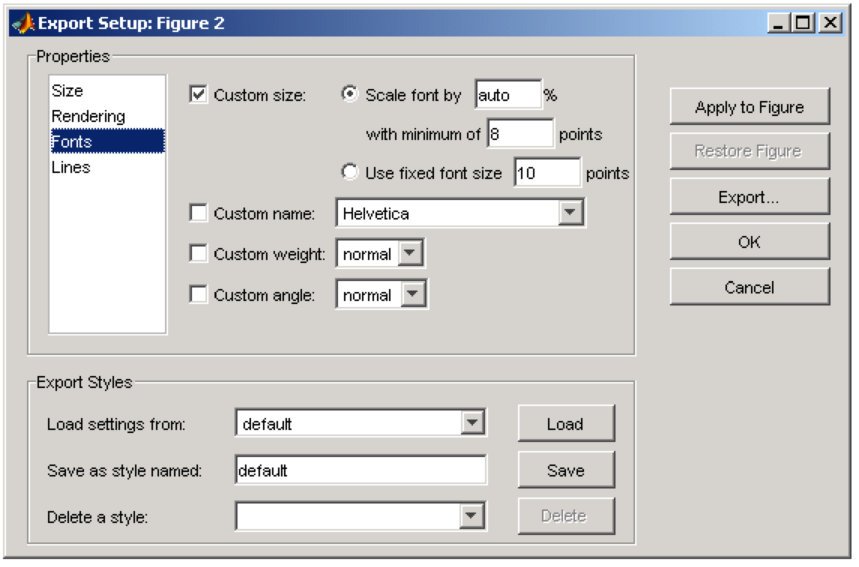
- Screen — The same resolution as used on your screen display
- A specific numeric setting — 150, 300, or 600 dpi
- auto — UNIX selects a suitable setting

Keep axis limits. Click the check box to keep axis tick marks and limits as shown. If unchecked, automatically adjust depending on figure size.

Show uicontrols. Click the check box to show all user interface controls in the figure. If unchecked, hide user interface controls.

Changing Font Characteristics

Click **Fonts** in the Export Setup dialog box to display this dialog box.



You can change the settings in this dialog box as follows:

Custom Size. Click the check box and use the radio buttons to select a relative or absolute font size for text in the figure.

- **Scale font by N %** — Increases or decreases the size of all fonts by a relative amount, N percent. Enter the word `auto` to automatically select the appropriate font size.
- **With minimum of N points** — You can specify a minimum font size when scaling the font by a percentage.
- **Use fixed font size N points** — Sets the size of all fonts to an absolute value, N points.

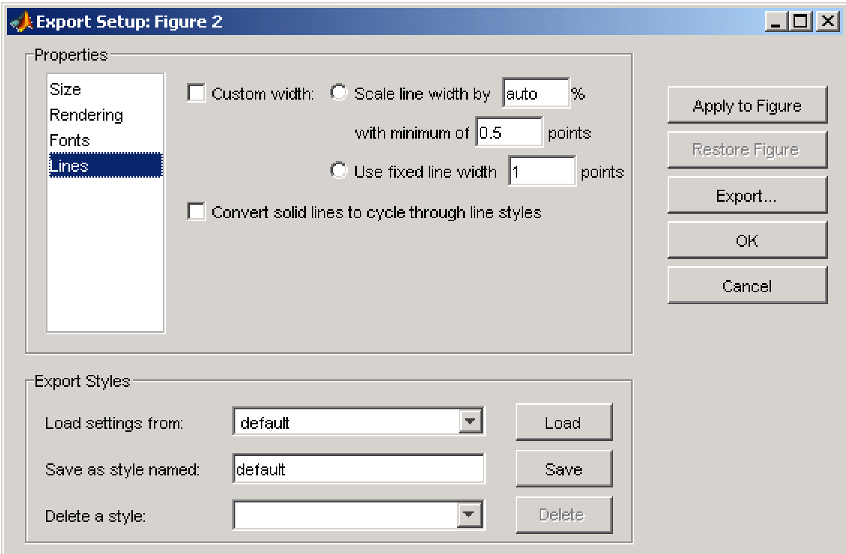
Custom Name. Click the check box and use the drop-down list to select a font name from those offered in the drop-down list.

Custom Weight. Click the check box and use the drop-down list to select the weight or thickness to be applied to text in the figure. Choose from `normal`, `light`, `demi`, or `bold`.

Custom Angle. Click the check box and use the drop-down list to select the angle to be applied to text in the figure. Choose from `normal`, `italic`, or `oblique`.

Changing Line Characteristics

Click **Lines** in the Export Setup dialog box to display this dialog box.



You can change the settings in this dialog box as follows:

Custom width. Click the check box and use the radio buttons to select a relative or absolute line size for the figure.

- **Scale line width by N %** — Increases or decreases the width of all lines by a relative amount, N percent. Enter the word `auto` to automatically select the appropriate line width.
- **With minimum of N points** — Specify a minimum line width when scaling the font by a percentage.
- **Use fixed line width N points** — Sets the width of all lines to an absolute value, N points.

Convert solid lines to cycle through line styles. When colored graphics are imported into an application that does not support color, lines that could formerly be distinguished by unique color are likely to appear the same. For example, a red line that shows an input level and a blue line showing output both appear as black when imported into an application that does not support colored graphics.

Clicking this check box causes exported lines to have different line styles, such as solid, dotted, or dashed lines rather than differentiating between lines based on color.

Saving and Loading Settings

If you think you might use these export settings at another time, you can save them now and reload them later. At the bottom of each Export Setup dialog box, there is a panel labeled **Export Styles**. To save your current export styles, type a name into the **Save as style named** text box, and then click **Save**.

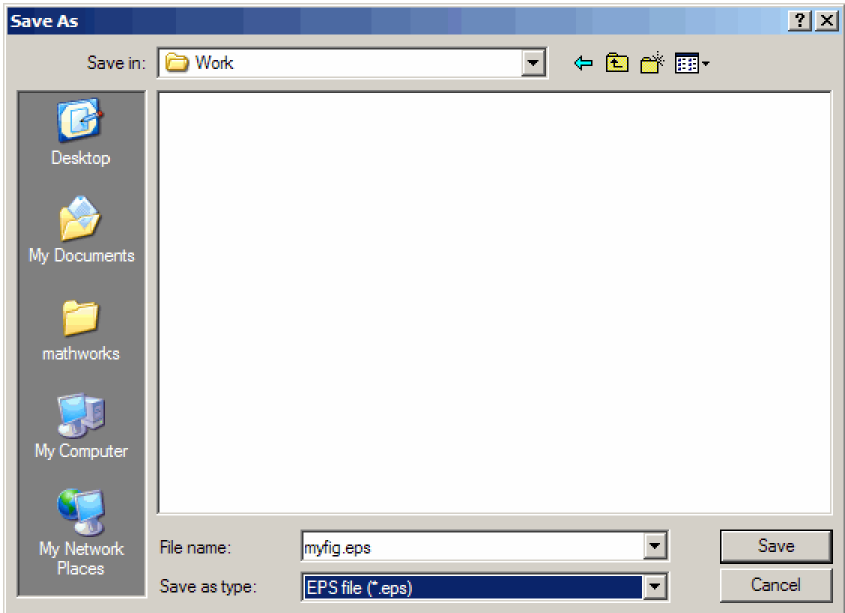
If you then click the **Load settings from** drop-down list, the name of the style you just saved appears among the choices of export styles you can load. To load a style, select one of the choices from this list and then click **Load**.

To delete any style you no longer have use for, select that style name from the **Delete a style** drop-down list and click **Delete**.

Exporting the Figure

When you finish setting the export style for your figure, you can export the figure to a file by clicking the **Export** button on the right side of any of the four Export Setup dialog boxes. As new window labeled **Save As** opens.

Select a directory to save the file in from the **Save in** list at the top. Select a file type for your file from the **Save as type** drop-down list at the bottom, and then enter a file name in the **File name** text box. Click the **Save** button to export the file.



For information on the graphics file formats supported by MATLAB, see "Choosing a Graphics Format" on page 7-73.

Exporting Using MATLAB Commands

Use the `print` function to print from the MATLAB command line or from a program. See "Printing and Exporting with print" on page 7-4 for basic information on printing from the command line.

To export the current or most recently active figure, type

```
print -dfileformat filename
```

where `fileformat` is a supported graphics format and `filename` is the name you want to give to the export file. MATLAB selects the filename extension, if you don't specify it.

You can also specify a number of options with the `print` function. These are shown in the Printing Options table on the `print` reference page.

For example, to export Figure No. 2 to file `spline2d.eps`, with 600 dpi resolution and using the EPS color graphics format, type

```
print -f2 -r600 -depsc spline2d
```

Graphics file formats are explained in more detail in the sections "Choosing a Graphics Format" on page 7-73 and "Description of Selected Graphics Formats" on page 7-80.

Exporting with getframe

You can use the `getframe` function with `imwrite` to export a graphic. `getframe` is often used in a loop to get a series of frames (figures) with the intention of creating a movie. No matter what the intrinsic resolution of the graphics might be, `getframe` only captures them at screen resolution.

Some of the benefits of using this export method over using `print` are

- You can use `getframe` to capture a portion of the figure, rather than the whole figure.
- `imwrite` offers greater flexibility for setting format-specific options, such as the bit depth and compression.

The drawbacks of using this method are

- `imwrite` uses built-in MATLAB formats only
- `getframe` and `imwrite` are limited to screen resolution

.Consequently, you do not have access to the Ghostscript formats available to you when exporting with the `print` function or **Export** menu.

How to Use getframe and imwrite. Use `getframe` to capture a figure and `imwrite` to save it to a file. `getframe` returns a structure containing the fields `cdata` and `colormap`. The `colormap` field is empty on true color displays. The following example captures the current figure and exports it to a PNG file.

```
I = getframe(gcf);
imwrite(I.cdata, 'myplot.png');
```

You should use the proper syntax of `imwrite` for the type of image captured. In the example above, the image is captured from a true color display. Because the `colormap` field is empty, it is not passed to `imwrite`.

Example — Exporting a Figure Using getframe and imwrite. This example offers device independence—it works for either RGB-mode or indexed-mode monitors.

```
X=getframe(gcf);
if isempty(X.colormap)
   imwrite(X.cdata, 'myplot.bmp')
else
   imwrite(X.cdata, X.colormap, 'myplot.tif')
end
```

For information about available file formats and format-specific options, see the `imwrite` reference page. For information about creating a movie from a series of frames, see the reference pages for `getframe` and `movie`, or see “Movies” on page 5-83 in Chapter 5, “Creating Specialized Plots”.

Saving Multiple Figures to an AVI File

You can also save multiple figures to an AVI file using the MATLAB `avifile` and `addframe` functions. AVI files can be used for animated sequences and do not need MATLAB to run, but do require an AVI viewer. For more information, see “Exporting Audio/Video Data” in the MATLAB Programming Fundamentals documentation.

Importing MATLAB Graphics into Other Applications

You can include MATLAB graphics in a wide variety of applications for word processing, slide preparation, modification by a graphics program,

presentation on the Internet, and so on. In general, the process is the same for all applications:

1. Use MATLAB graphics to create the figure you want to import into another application.

2. Export the MATLAB figure to one of the supported graphics file formats, selecting a format that is both appropriate for the type of figure and supported by the target application. See "Choosing a Graphics Format" on page 7-73 for help.

3. Use the import features of the target application to import the graphics file.

Edit Before You Export. Vector graphics may be fully editable in a few high-end applications, but most applications do not support editing beyond simple resizing. Bitmaps cannot be edited with quality results unless you use a software package devoted to image processing. In general, you should try to make all the necessary settings while your figure is still in MATLAB.

Importing into Microsoft Applications. To import your exported figure into a Microsoft application, select **Picture** from the **Insert** menu. Then select **From File** and navigate to your exported file. If you use the clipboard to perform your export operations, you can take advantage of the recommended MATLAB settings for Microsoft Word and PowerPoint®.

Example — Importing an EPS Graphic into LaTeX. This example shows how to import an EPS file named `peaks.eps` into LaTeX.

```
\documentclass{article}

\usepackage{graphicx}

\begin{document}

\begin{figure}[h]
\centerline{\includegraphics[height=10cm]{peaks.eps}}
\caption{Surface Plot of Peaks}
\end{figure}

\end{document}
```

EPS graphics can be edited after being imported to LaTeX. For example, you can specify the height in any LaTeX-compatible dimension. To set the height to 3.5 inches, use the command

```
height=3.5in
```

You can use the `angle` function to rotate the graph. For example, to rotate the graph 90 degrees, add

```
angle=90
```

to the same line of code that sets the height, i.e., `[height=10cm,angle=90]`.

Exporting to the Windows or Macintosh Clipboard

You can export a figure to the Windows or Macintosh clipboard. The formats used are discussed below.

Windows Clipboard Format

Either of two graphics formats are used to write clipboard data on Windows: EMF color vector or BMP 8-bit color bitmap.

By default, the graphics format is automatically selected for you, based on the rendering method used to display the figure. For figures rendered with OpenGL® or Z-buffer, MATLAB uses the BMP format. For figures rendered with Painter's, the EMF format is used. For information about how rendering methods are chosen, see "The Default MATLAB Renderer" on page 7-55.

To override the autmatic selection, specify the format of your choice using either the Windows Copy Options Preferences dialog box, or the `-d` switch in the print command.

Macintosh Clipboard Format

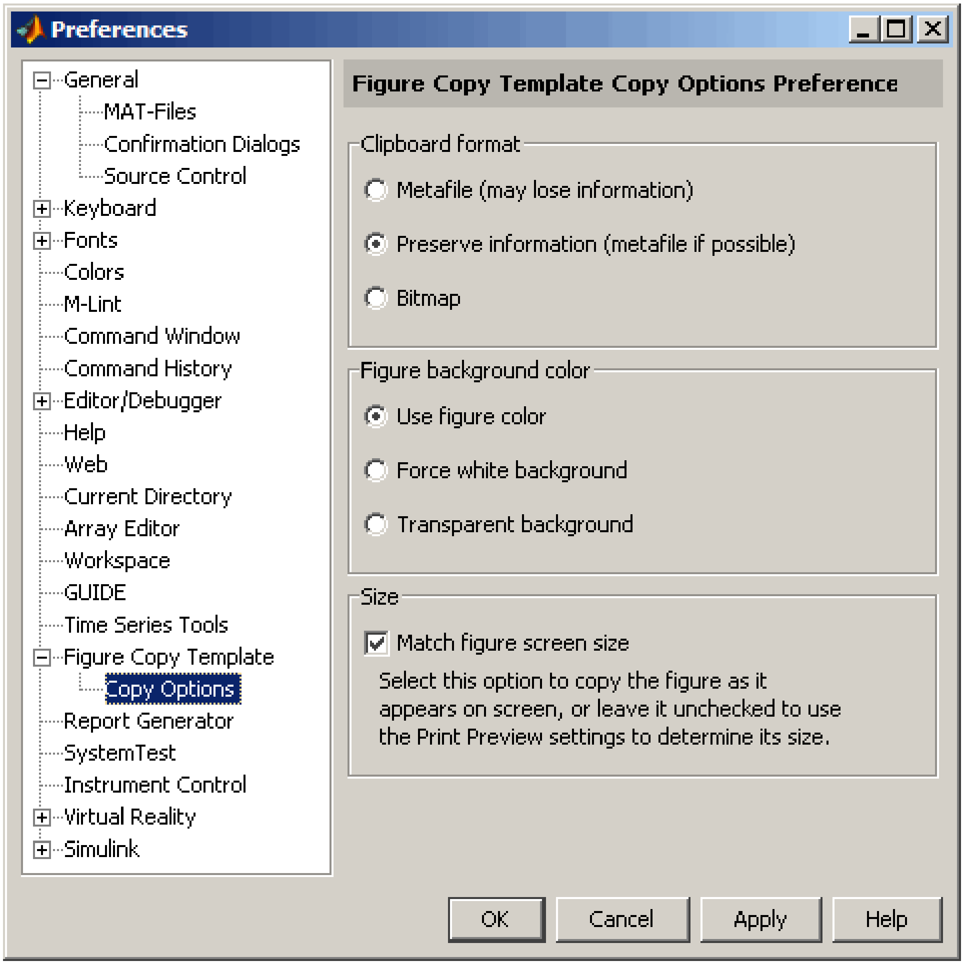
On Macintosh platforms (using Java™ figures, the default), clipboard data are always written as RGB truecolor bitmaps. The entire figure window is captured.

Exporting to the Clipboard Using GUIs

Before you export the figure to the clipboard, you can use the Copy Options Preferences dialog box to select a nondefault graphics format, or to adjust certain figure settings. These settings become the new defaults for all figures exported to the clipboard.

Note When exporting to the clipboard in Windows metafile format (e.g., `print -dmeta`), the settings from the figure Copy Options Preferences template are ignored.

To open the Copy Options Preferences dialog box, select **Copy Options** from the figure window's **Edit** menu. Any changes you make with this dialog box affect only the clipboard copy of the figure; they do not affect the way the figure looks on the screen.



Settings you can change in the Copy Options Preferences dialog box are as follows:

Clipboard format. To copy the figure in EMF color vector format, select **Metafile**. To automatically select the format for you, select **Preserve information**. To use BMP 8-bit color bitmap format, select **Bitmap**. MATLAB clipboard data uses the metafile format whenever possible.

Note On Macintosh platforms, the **Copy Options** dialog box does not have the **Clipboard format** options.

Figure background color. To keep the background color the same as it appears on the screen, select **Use figure color**. To make the background white, select **Force white background**. For a background that is transparent, for example, a slide background to frame the axes part of a figure, select **Transparent background**.

Size. Select **Match figure screen size** to copy the figure as it appears on the screen, or leave it unselected to use the **Width** and **height** options in the Export Setup dialog to determine its size.

1 Open the Copy Options Preferences dialog box if you need to make any changes to those preferences used in copying to the clipboard.

2 Click **OK** to see the new preferences. These will be used for all future figures exported to the clipboard.

3 Select **Copy Figure** from the figure window's **Edit** menu to copy the figure to the clipboard.

Exporting to the Windows or Macintosh Clipboard Using MATLAB Commands

Export to the clipboard using the `print` function with a graphics format, but no filename. You must use one of the following clipboard formats: `-dbitmap` or `-dmeta`. These switches create a Windows bitmap (BMP) or an enhanced metafile (EMF), respectively.

For example, to export the current figure to the clipboard in enhanced metafile format, type

```
print -dmeta
```

Note When printing, the `print -d` option specifies a printer driver. When exporting, the `print -d` option specifies a graphics format.

Examples of Printing and Exporting

In this section...

Printing a Figure at Screen Size

By default, your figure prints at 8-by-6 inches. This size includes the area delimited by the background. This example shows how to print or export your figure the same size it is displayed on your screen.

Using the Graphical User Interface

1. Resize your figure window to the size you want it to be when printed.
2. Select **Print Preview** from the figure window's **File** menu, and select the **Layout** tab.
3. In the **Placement** panel, select **Auto (Actual Size, Centered)**.
4. Click **Print** in the upper right corner to print the figure.
5. The Print dialog box opens for you to print the figure.

Using MATLAB Commands

Set the `PaperPositionMode` property to `auto` before printing the figure.

```
set(gcf, 'PaperPositionMode', 'auto');
print
```

If later you want to print the figure at its original size, set `PaperPositionMode` back to `'manual'`.

Printing with a Specific Paper Size

The MATLAB default paper size is 8.5-by-11 inches. This example shows how to change the paper size to 8.5-by-14 inches by selecting a paper type (Legal).

Using the Graphical User Interface

1. Select **Print Preview** from the figure window's **File** menu, and select the **Layout** tab.
2. Select the `Legal` paper type from the list in the **Paper** panel. The `Width` and `Height` fields update to `8.5` and `14`, respectively.
3. Make sure that **Units** is set to `inches`.
4. Click **Print** in the upper right corner to print the figure.
5. The Print dialog box opens for you to print the figure.

Using MATLAB Commands

Set the `PaperUnits` property to `inches` and the `PaperType` property to `Legal`.

```
set(gcf, 'PaperUnits', 'inches');
set(gcf, 'PaperType', 'Legal');
```

Alternatively, you can set the `PaperSize` property to the size of the paper, in the specified units.

```
set(gcf, 'PaperUnits', 'inches');
set(gcf, 'PaperSize', [8.5 14]);
```

Printing a Centered Figure

This example sets the size of a figure to 5.5-by-3 inches and centers it on the paper.

Using the Graphical User Interface

1. Select **Print Preview** from the figure window's **File** menu, and select the **Layout** tab.
2. Make sure **Use manual size and position** is selected.
3. Enter `5.5` in the **Width** field and `3` in the **Height** field.
4. Make sure that **Units** field is set to `inches`.
5. Click **Center**.
6. Click **OK**.
7. Click **Print** to open the Print dialog box and print the figure.

Using MATLAB Commands

1. Start by setting `PaperUnits` to `inches`.

```
set(gcf, 'PaperUnits', 'inches')
```

2. Use `PaperSize` to return the size of the current paper.

```
papersize = get(gcf, 'PaperSize')

papersize =
              8.5000   11.0000
```

3. Initialize variables to the desired width and height of the figure.

```
width = 5.5;         % Initialize a variable for width.
height = 3;          % Initialize a variable for height.
```

4. Calculate a left margin that centers the figure horizontally on the paper. Use the first element of `papersize` (width of paper) for the calculation.

```
left = (papersize(1)- width)/2

left =
          1.5000
```

5 Calculate a bottom margin that centers the figure vertically on the paper. Use the second element of `papersize` (height of paper) for the calculation.

```
bottom = (papersize(2)- height)/2

bottom =
         4
```

6 Set the figure size and print.

```
myfiguresize = [left, bottom, width, height];
set(gcf, 'PaperPosition', myfiguresize);
print
```

Exporting in a Specific Graphics Format

Export a figure to a graphics-format file when you want to import it at a later time into another application such as a word processor.

Using the Graphical User Interface

1. Select **Save As** from the figure window's **File** menu.
2. Use the **Save in** field to navigate to the directory in which you want to save your file.
3. Select a graphics format from the **Save as type** list.
4. Enter a filename in the **File name** field. An appropriate file extension, based on the format you chose, is displayed.
5. Click **Save** to export the figure.

Using MATLAB Commands

From the command line, you must specify the graphics format as an option. See the `print` reference page for a complete list of graphics formats and their corresponding option strings.

This example exports a figure to an EPS color file, `myfigure.eps`, in your current directory.

```
print -depsc myfigure
```

This example exports Figure No. 2 at a resolution of 300 dpi to a 24-bit JPEG file, `myfigure.jpg`.

```
print -djpeg -f2 -r300 myfigure
```

This example exports a figure at screen size to a 24-bit TIFF file, `myfigure.tif`.

```
set(gcf, 'PaperPositionMode', 'auto')   % Use screen size
print -dtiff myfigure
```

Exporting in EPS Format with a TIFF Preview

Use the `print` function to export a figure in EPS format with a TIFF preview. When you import the figure, the application can display the TIFF preview in the source document. The preview is color if the exported figure is color, and black and white if the exported figure is black and white.

This example exports a figure to an EPS color format file, `myfigure.eps`, and includes a color TIFF preview.

```
print -depsc -tiff myfigure
```

This example exports a figure to an EPS black-and-white format file, `myfigure.eps`, and includes a black-and-white TIFF preview.

```
print -deps -tiff myfigure
```

Exporting a Figure to the Clipboard

Export a figure to the clipboard in graphics format when you want to paste it into another Windows or Macintosh application such as a word processor.

Using the Graphical User Interface

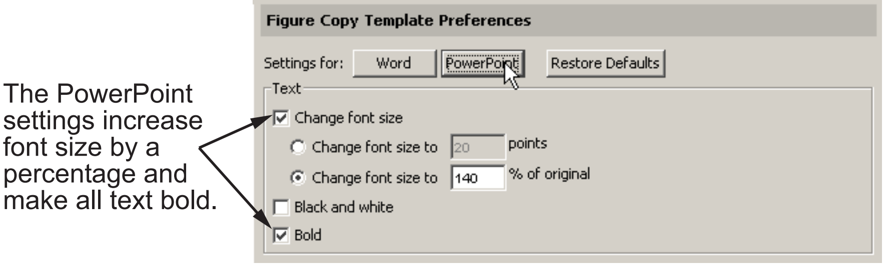
This example exports a figure to the clipboard in enhanced metafile (EMF) format. Figure settings are chosen that would make the exported figure suitable for use in a Microsoft Word or PowerPoint slide. Changing the settings modifies the figure displayed on the screen.

1 Create a figure containing text. You can use the following code.

```
x = -pi:0.01:pi;
h = plot(x, sin(x));
title('Sine Plot');
```

2 Select **Preferences** from the **File** menu of either the figure or main desktop window. Then select **Figure Copy Template** from the Preferences dialog box.

3 In the **Figure Copy Template Preferences** panel, click the **PowerPoint** button. The suggested settings for PowerPoint are added to the template.



The Text Panel after pushing the **PowerPoint** button

4 In the **Lines** panel, change the **Custom width** to 4 points.

5 In the **Uicontrols and axes** panel, select **Keep axes limits and tick spacing** to prevent tick marks and limits from possibly being rescaled when you export.

6 Click **Apply to Figure**. The changes appear in the figure window.

If you don't like the way your figure looks with the new settings, restore it to its original settings by clicking the **Restore Figure** button.

7 In the left pane of the Preferences dialog box, expand the **Figure Copy Template** topic. Select **Copy Options**.

8 In the **Copy Options** panel, select **Metafile** to export the figure in EMF format.

9 Check that **Transparent background** is selected. This choice makes the figure background transparent and allows the slide background to frame the axes part of the figure.

10 Clear the **Match figure screen size** check box so that you can use your own figure size settings.

11 Click **OK**.

12 Select **Export Setup** from the figure window's **File** menu.

13 Select the **Size** properties, and set **Width** to `6` and **Height** to `4.5`. Make sure that **Units** are set to `inches`.

14 Click **Close**.

15 Select **Copy Figure** from the **Edit** menu. Your figure is now exported to the clipboard and can be pasted into another Windows application, such as PowerPoint.

Using MATLAB Commands

Use the `print` function and one of two clipboard formats (`-dmeta`, `-dbitmap`) to export a figure to the clipboard. Do *not* specify a filename.

This example exports a figure to the clipboard in enhanced metafile (EMF) format.

```
print -dmeta
```

This example exports a figure to the clipboard in bitmap (BMP) 8-bit color format.

```
print -dbitmap
```

Changing a Figure's Settings

In this section...

Parameters that Affect Printing

The table below shows parameters that you can set before submitting your figure to the printer.

The Parameter column lists all parameters that you can change.

The Default column shows the MATLAB default setting.

The Dialog Box column shows which dialog box to use to set that parameter. If you can make this setting on only one platform, this is noted in parentheses: (W) for Windows, and (U) for UNIX.

Some dialog boxes have tabs at the top to enable you to select a certain category. These categories are denoted in the table below using the format `<dialogbox>/<tabname>`. For example, **Print Preview/Layout...** in this column means to use the Print Preview dialog box, selecting the **Layout** tab.

The `print` Command or `set` Property column shows how to set the parameter using the MATLAB `print` or `set` function. When using `print`, the table shows the appropriate command option (for example, `print -loose`). When using `set`, it shows the property name to set along with the type of object (for example, (Line) for line objects).

Parameter	Default	Dialog Box	`print` Command or `set` Property
Select figure	Last active window	None	`print -f`*`handle`*
Select printer	System default	**Print**	`print -p`*`printer`*
Figure size	8-by-6 inches	**Print Preview/Layout**	`PaperSize` (Figure) `PaperUnits` (Figure)
Position on page	0.25 in. from left, 2.5 in. from bottom	**Print Preview/Layout**	`PaperPosition` (Figure) `PaperUnits` (Figure)
Position mode	Manual	**Print Preview/Layout**	`PaperPositionMode` (Figure)
Paper type	Letter	**Print Preview/Layout**	`PaperType` (Figure)
Paper orientation	Portrait	**Print Preview/Layout**	`PaperOrientation` (Figure)
Renderer	Selected automatically	**Print Preview/Advanced**	`print -zbuffer | -painters | -opengl`
Renderer mode	Auto	**Print Preview/Advanced**	`RendererMode` (Figure)
Resolution	Depends on driver or graphics format	**Print Preview/Advanced**	`print -r`*`resolution`*
Axes tick marks	Recompute	**Print Preview/Advanced**	`XTickMode, etc.` (Axes)

Parameter	Default	Dialog Box	`print` Command or `set` Property
Background color	Force to white	**Print Preview/Color**	`Color` (Figure) `InvertHardCopy` (Figure)
Font size	As in the figure	**Print Preview/Lines/Text**	`FontSize` (Text)
Bold font	Regular font	**Print Preview/Lines/Text**	`FontWeight` (Text)
Line width	As in the figure	**Print Preview/Lines/Text**	`LineWidth` (Line)
Line style	Black or white	**Figure Copy Template**	`LineStyle` (Line)
Line and text color	Black and white	**Print Preview/Lines/Text**	`Color` (Line, Text)
CMYK color	RGB color	**Print Preview/Color** (U)	`print -cmyk`
UI controls	Printed	**Print Preview/Advanced**	`print -noui`
Bounding box	Tight	N/A	`print -loose`
Copy background	Transparent	**Copy Options** (W)	See "Background color"
Copy size	Same as screen size	**Copy Options** (W)	See "Figure Size"

Selecting the Figure

By default, the current figure prints. If you have more than one figure open, the current figure is the last one that was active. To make a different figure active, click it to bring it to the foreground.

Using MATLAB Commands

Specify a figure handle using the command

```
print -fhandle
```

This example sends Figure No. 2 to the printer. A figure's number is usually its handle.

```
print -f2
```

Selecting the Printer

You can select the printer you want to use with the Print dialog box or with the `print` function.

Using the Graphical User Interface

1 Select **Print** from the figure window's **File** menu.

2 Select the printer from the list box near the top of the Print dialog box.

3 Click **OK**.

Using MATLAB Commands

You can select the printer using the `-P` switch of the `print` function.

This example prints Figure No. 3 to a printer called `Calliope`.

```
print -f3 -PCalliope
```

If the printer name has spaces in it, put single quotation marks around the `-P` option, as shown here.

```
print '-Pmy local printer'
```

Using a Network Print Server. On Windows systems, you can print to a network print server using the form shown here for a printer named `trinity` located on a computer named `PRINTERS`.

```
print -P\\PRINTERS\trinity
```

Note On Windows platforms, when you use the `-P` option to identify a printer to use, if you specify any driver other than `-dwin` or `-dwinc`, MATLAB output goes to a file with an appropriate extension but does not send it to the printer; you can then copy that file to a printer.

Setting the Figure Size and Position

The default output figure size is 8 inches wide by 6 inches high, which maintains the aspect ratio (width to height) of the MATLAB figure window. The figure's default position is centered both horizontally and vertically when printed to a paper size of 8.5-by-11 inches.

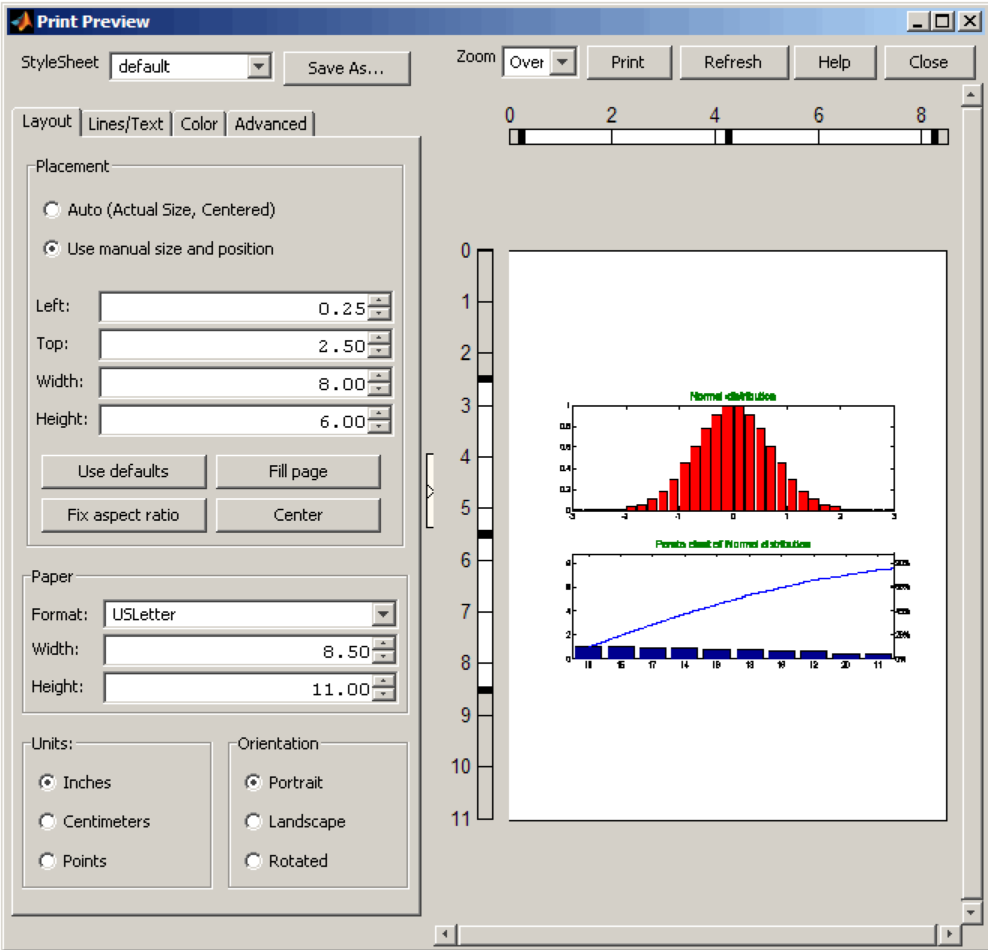
You can change the size and position of the figure:

- "Using the Graphical User Interface" on page 7-47
- "Using MATLAB Commands" on page 7-49

Using the Graphical User Interface

Select **Print Preview** from the figure window's **File** menu to open the Print Preview dialog box. Click the **Layout** tab to make changes to the size and position of your figure on the printed page.

Use the text edit boxes on the left to enter new dimensions for your figure. Or, use the handlebars on the rulers in the right-hand pane to drag the margins and location of your figure with the mouse. The outer handlebars move the figure toward or away the nearest margin, while the central handlebar repositions the figure on the page without changing its proportions. Guidelines appear while you are using the handlebars.



Settings you can change in the **Layout** tab are as follows:

Placement. Choose whether you want the figure to be the same size as it is displayed on your screen, or you want to manually change its size using the options in the **Layout** pane.

When you select the **Use manual size and position** mode, type the widths of any of the four margins and the preview image responds after each entry you

make. Select units of measure (inches/centimeters/points) with pushbuttons on the **Units** section on the bottom of the pane.

You can use the four buttons at the bottom of the Placement section to expand the figure to fill the page, make its aspect ratio (ratio of y-extent to x-extent) as printed match that of the figure, center the figure on the page, or restore the setup to what it was when you opened the Print Preview dialog. Selecting **Fill page** can alter the aspect ratio of your image. To get the maximum figure size without altering the aspect ratio, select **Fix aspect ratio**.

Auto (actual size, centered). Select this option to center the figure on the page; it will be the same size as it is in the figure window. The four buttons below the control are dimmed when you select this option.

Note Changes you make using Print Preview affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

To print your figure with a specific size or position, make sure the `PaperPositionMode` property is set to `manual` (the default). Then set the `PaperPosition` property to the desired size and position.

The `PaperPosition` property references a four-element row vector that specifies the position and dimensions of the printed output. The form of the vector is

```
[left bottom width height]
```

where

- `left` specifies the distance from the left edge of the paper to the left edge of the figure.
- `bottom` specifies the distance from the bottom of the paper to the bottom of the figure.
- `width` and `height` specify the figure's width and height.

The default values for `PaperPosition` are

```
[0.25 2.5 8.0 6.0]
```

This example sets the figure size to a width of 4 inches and height of 2 inches, with the origin of the figure positioned 2 inches from the left edge of the paper and 1 inch from the bottom edge.

```
set(gcf, 'PaperPositionMode', 'manual');
set(gcf, 'PaperUnits', 'inches');
set(gcf, 'PaperPosition', [2 1 4 2]);
```

Note `PaperPosition` specifies a bottom margin, rather than a top margin as Print Preview does. When you set the top margin using Print Preview, This setting is used to calculate the bottom margin, and updates the `PaperPosition` property appropriately.

Setting the Paper Size or Type

Set the paper size by specifying the dimensions or by choosing from a list of predefined paper types. If you do not set a paper size or type, the default paper size of 8.5-by-11 inches is used.

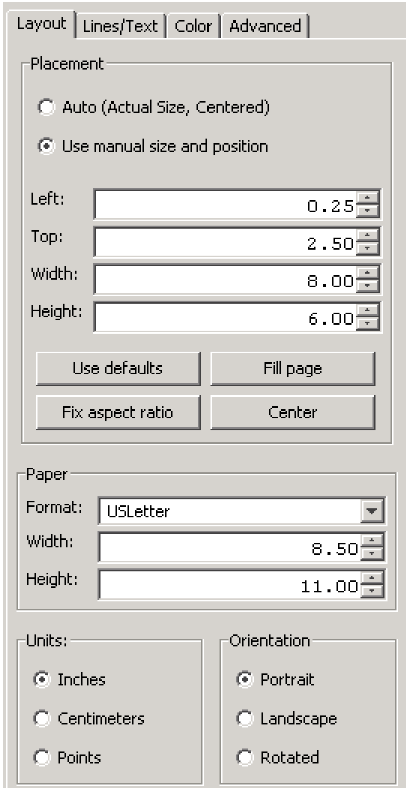
Paper-size and paper-type settings are interrelated—if you set a paper type, the paper size also updates. For example, if you set the paper type to `US Legal`, the width of the paper updates to 8.5 inches and the height to 14 inches.

You can change the paper size and orientation:

- “Using the Graphical User Interface” on page 7-50
- “Using MATLAB Commands” on page 7-52

Using the Graphical User Interface

Select **Print Preview** from the figure window’s **File** menu to open the Print Preview dialog box. Click the **Layout** tab to make changes to the paper type and orientation of the figure on the printed page.



Settings you can change in the **Layout** tab are as follows:

Paper Format, Units and Orientation. Select a paper type from the list under **Format**. If there is no paper type with suitable dimensions, enter your own dimensions in the **Width** and **Height** fields. Make sure **Units** is set appropriately to `inches`, `centimeters`, or `points`. If you change units after setting a paper width and height, the **Width** and **Height** fields update to use the units you just selected. The page region in the preview pane updates to show the new paper format or size when you change them.

Use the **Orientation** buttons to select how you want the figure to be oriented on the printed page. The illustration under "Setting the Paper Orientation" on page 7-52 shows the three types of orientation you can choose from.

Note Changes you make using Print Preview affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

Set the `PaperType` property to one of the built-in MATLAB paper types, or set the `PaperSize` property to the dimensions of the paper.

When you select a paper type, the unit of measure is not automatically updated. We recommend that you set the `PaperUnits` property first.

For example, these commands set the units to `centimeters` and the paper type to `A4`.

```
set(gcf, 'PaperUnits', 'centimeters');
set(gcf, 'PaperType', 'A4');
```

This example sets the units to `inches` and sets the paper size of 5-by-7 inches.

```
set(gcf, 'PaperUnits', 'inches');
set(gcf, 'PaperSize', [5 7]);
```

If you set a paper size for which there is no matching paper type, the `PaperType` property is automatically set to `'custom'`.

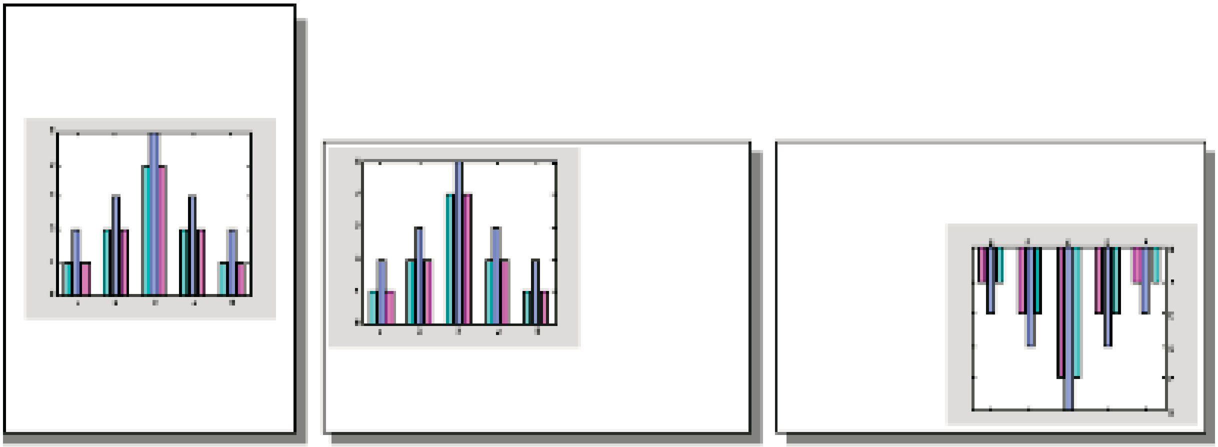
Setting the Paper Orientation

Paper orientation refers to how the paper is oriented with respect to the figure. The choices are **Portrait** (the default), **Landscape**, and **Rotated**.

You can change the orientation of the figure:

- "Using the Graphical User Interface" on page 7-53
- "Using MATLAB Commands" on page 7-53

The figure below shows the same figure printed using the three different orientations.



Portrait Landscape Rotated (by 180 degrees)

Note The **Rotated** orientation is not supported by all printers. When the printer does not support it, landscape is used.

Using the Graphical User Interface

1. Select **Print Preview** from the figure window's **File** menu and select the **Layout** tab. (See "Using the Graphical User Interface" on page 7-50).

2. Select the appropriate option button under **Orientation**.

3. Click **Close**.

Using MATLAB Commands

Use the `PaperOrientation` figure property or the `orient` function. Use the `orient` function if you always want your figure centered on the paper.

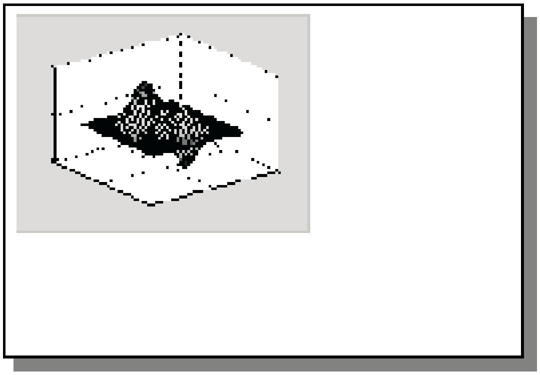
The following example sets the orientation to `landscape`:

```
set(gcf, 'PaperOrientation', 'landscape');
```

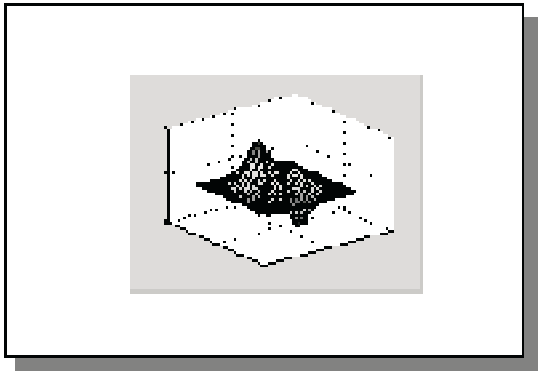
Centering the Figure. If you set the PaperOrientation property from portrait to either of the other two orientation schemes, you might find that what was previously a centered image is now positioned near the paper's edge. You can either adjust the position (use the PaperPosition property), or you can use the orient function, which always centers the figure on the paper.

The orient function takes the same argument names as PaperOrientation. For example,

```
orient rotated;
```



Orientation set to 'landscape' using 'PaperOrientation' property.



Orientation set to 'landscape' using orient function.

Selecting a Renderer

A renderer is software and/or hardware that processes graphics data (such as vertex coordinates) to display, print, or export a figure. You can change the renderer from the one used to draw a figure to another renderer when printing it:

- "Using the Graphical User Interface" on page 7-56
- "Using MATLAB Commands" on page 7-57

Supported Renderers

MATLAB supports three rendering methods with the following characteristics:

Painter's

- Draws figures using vector graphics
- Generally produces higher resolution results
- The fastest renderer when the figure contains only simple or small graphics objects
- The only renderer possible when printing with the HPGL print driver or exporting to an Adobe® Illustrator file
- The best renderer for creating PostScript or EPS files
- Cannot render figures that use RGB color for patch or surface objects
- Does not show lighting or transparency

Z-buffer

- Draws figures using bitmap (raster) graphics
- Faster and more accurate than Painter's
- Can consume a lot of system memory when displaying a complex scene
- Shows lighting, but not transparency

OpenGL

- Draws figures using bitmap (raster) graphics
- Generally faster than Painter's or Z-buffer
- Can access graphics rendering hardware available on some systems
- Shows both lighting and transparency

For more detailed information about the rendering methods, see `Renderer` on the Figure Properties reference page.

The Default MATLAB Renderer

By default, OpenGL tries to optimize the rendering method based on the attributes of the figure (its complexity and the settings of various Handle Graphics properties) and in some cases, the printer driver or file format used.

In general, renderers are selected as follows:

- Painter's, for line plots, area plots (bar graphs, histograms, etc.), and simple surface plots
- Z-buffer, when the computer screen is not true color or when the `opengl` function was called with `selection_mode` set to `neverselect`
- OpenGL, for complex surface plots using interpolated shading and any figure using lighting

The `RendererMode` property describes whether to automatically select the renderer based on the contents of the figure (when set to `auto`), or to use the `Renderer` property that you have indicated (when set to `manual`).

Reasons for Manually Setting the Renderer

Two reasons to set the renderer yourself are

- To make your printed or exported figure look the same as it did on the screen. The rendering method used for printing and exporting the figure is not always the same method used to display the figure.
- To avoid unintentionally exporting your figure as a bitmap within a vector format. For example, high-complexity MATLAB plots typically render using OpenGL or Z-buffer. If you export a high-complexity figure to the EPS or EMF vector formats without specifying a rendering method, either OpenGL or Z-buffer might be used, each of which creates bitmap graphics.

  Storing a bitmap in a vector file can generate a very large file that takes a long time to print. If you use one of these formats and want to make sure that your figure is saved as a vector file, be sure to set the rendering method to Painter's.

Using the Graphical User Interface

1. Open the Print Preview dialog box by selecting **Print Preview** from the figure window's **File** menu. Select the **Advanced** tab.

2. In the `Renderer` drop-down menu, select the desired rendering method from the list box.

3 Click **Close**.

Using MATLAB Commands

You can use the `Renderer` property or a switch with the `print` function to set the renderer for printing or exporting. These two lines each set the renderer for the current figure to Z-buffer.

```
set(gcf, 'Renderer', 'zbuffer');
```

or

```
print -zbuffer
```

The first example saves the new value of `Renderer` with the figure; the second example only affects the current print or export operation.

When you set the `Renderer` property, the `RendererMode` property is automatically reset from `auto` (the factory default) to `manual`.

Setting the Resolution

Resolution refers to how accurately your figure is rendered when printed or exported. Higher resolutions produce higher quality output. The specific definition of resolution depends on whether your figure is output as a bitmap or as a vector graphic.

You can change the resolution used to print a figure:

- "Using the Graphical User Interface" on page 7-59
- "Using the Graphical User Interface on UNIX Platforms" on page 7-71
- "Using MATLAB Commands" on page 7-59

Default Resolution and When You Can Change It

The default resolution depends on the renderer used and the graphics format or printer driver specified. The following two tables summarize the default resolutions and whether you can change them.

Resolutions Used with Graphics Formats

Graphics Format	Default Resolution	Can Be Changed?
Built-in MATLAB export formats, (except for EMF, EPS, and ILL)	150 dpi (always use OpenGL or Z-buffer)	Yes
EMF export format (Enhanced Metafile)	150 dpi	Yes
EPS (Encapsulated PostScript)	150 dpi, if OpenGL or Z-buffer; 864 dpi if Painter's	Yes
ILL export format (Adobe Illustrator)	72 dpi (always uses Painter's)	No
Ghostscript export formats	72 dpi (always uses OpenGL or Z-buffer)	No

Resolutions Used with Printer Drivers

Printer Driver	Default Resolution	Can Be Changed?
Windows and PostScript drivers	150 dpi, if OpenGL or Z-buffer; 864 dpi if Painter's	Yes
Ghostscript driver	150 dpi, if OpenGL or Z-buffer; 864 dpi if Painter's	Yes
HPGL driver	1116 dpi (always uses Painter's)	Yes

Choosing a Setting

You might need to determine your resolution requirements through experimentation, but you can also use the following guidelines.

For Printing. The default resolution of 150 dpi is normally adequate for typical laser-printer output. However, if you are preparing figures for high-quality printing, such as a textbook or color brochures, you might want to use 200 or 300 dpi. The resolution you can use can be limited by the printer's capabilities.

For Exporting. If you are exporting your figure, base your decision on the resolution supported by the final output device. For example, if you will import your figure into a word processing document and print it on a printer that supports a maximum resolution setting of 300 dpi, you could export your figure using 300 dpi to get a precise one-to-one correspondence between pixels in the file and dots on the paper.

Note The only way to set resolution when exporting is with the `print` function.

Impact of Resolution on Size and Memory Needed

Resolution affects file size and memory requirements. For both printing and exporting, the higher the resolution setting, the longer it takes to render your figure.

Using the Graphical User Interface

To set the resolution for built-in MATLAB printer drivers:

1. From the Print dialog box, click **Properties**. This opens a new dialog box. (This box can differ from one printer to another.)

2. You may be able to set the resolution from this dialog. If not, then click **Advanced** to get to a dialog box that enables you to do this.

3. Set the resolution, and then click **OK**. (The resolution setting might be labeled by another name, such as "Print Quality.")

Using MATLAB Commands

If you use a Windows printer driver, you can only set the resolution using the Windows Document Properties dialog box.

Otherwise, to set the resolution for printing or exporting, the syntax is

```
print -rnumber
```

where `number` is the number of dots per inch. To print or export a figure using screen resolution, set `number` to `0` (zero).

This example prints the current figure with a resolution of 100 dpi:

```
print -r100
```

This example exports the current figure to a TIFF file using screen resolution:

```
print -r0 -dtiff myfile.tif
```

Setting the Axes Ticks and Limits

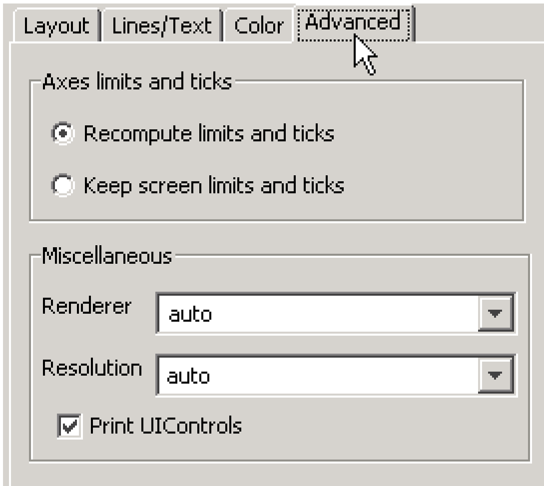
The default output size, 8-by-6 inches, is normally larger than the screen size. If the size of your printed or exported figure is different from its size on the screen, the number and placement of axes tick marks scale to suit the output size. This section shows you how to lock them so that they are the same as they were when displayed.

You can change the resolution used when printing a MATLAB figure:

- “Using the Graphical User Interface” on page 7-61
- “Using MATLAB Commands” on page 7-62

Using the Graphical User Interface

Select **Print Preview** from the figure window's **File** menu to open the Print Preview dialog box. Select the **Advanced** tab to make changes to the axes, UI controls, or renderer settings.



Settings you can change in the **Advanced** tab are as follows, by panel:

Axes limits and ticks. If the size of your printed or exported figure is different from its size on the screen, the number and placement of axes tick marks scale to suit the output size. Select **Keep screen limits and ticks** to lock them so that they are the same as they were when displayed. If you want to automatically adjust the ticks and limits when scaling for printing, select **Recompute limits and ticks**.

Miscellaneous. Use the **Renderer** drop-down menu to specify which renderer to use in printing the figure. Set the renderer to `Painters`, `Z-buffer`, or `OpenGL`, or select `auto` to automatically decide which one to use, depending on the characteristics of the figure. (See "Selecting a Renderer" on page 7-54).

Use the **Resolution** drop-down menu to specify the resolution, in dots per inch (DPI), at which to render and print the figure. You can select `150`, `300`, or `600` DPI, or type in a different value (positive integer).

Figure UI Controls. By default, user interface controls are included in your printed or exported figure. Clear the **Print UIControls** check box to exclude them. (See "Excluding User Interface Controls form Printed Output" on page 7-71).

Note Changes you make using Print Preview affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

To set the `XTickMode`, `YTickMode`, and `ZTickMode` properties to `manual`, type

```
set(gca, 'XTickMode', 'manual');
set(gca, 'YTickMode', 'manual');
set(gca, 'ZTickMode', 'manual');
```

Setting the Background Color

You can keep the background the same as is shown on the screen when printed, or change the background to white. There are two types of background color settings in a figure: the axes background and the figure background. The default displayed color of both backgrounds is gray, but you can set them to any of several colors.

Regardless of the background colors in your displayed figure, by default, they are always changed to white when you print or export. This section shows you how to retain the displayed background colors in your output.

Using the Graphical User Interface

To retain the background color on a per figure basis:

1. Open the Print Preview dialog box by selecting **Print Preview** from the figure window's **File** menu. Select the **Color** tab.

2. Select **Same as figure**.

3. Click **Close**.

If you are exporting your figure using the clipboard, use the **Copy Options** panel of the Preferences dialog box.

Using MATLAB Commands

To retain your background colors, use

```
set(gcf, 'InvertHardCopy', 'off');
```

The following example sets the figure background color to `blue`, the axes background color to `yellow`, and then sets `InvertHardCopy` to `off` so that these colors appear in your printed or exported figure.

```
set(gcf, 'color', 'blue');
set(gca, 'color', 'yellow');
set(gcf, 'InvertHardCopy', 'off');
```

Setting Line and Text Characteristics

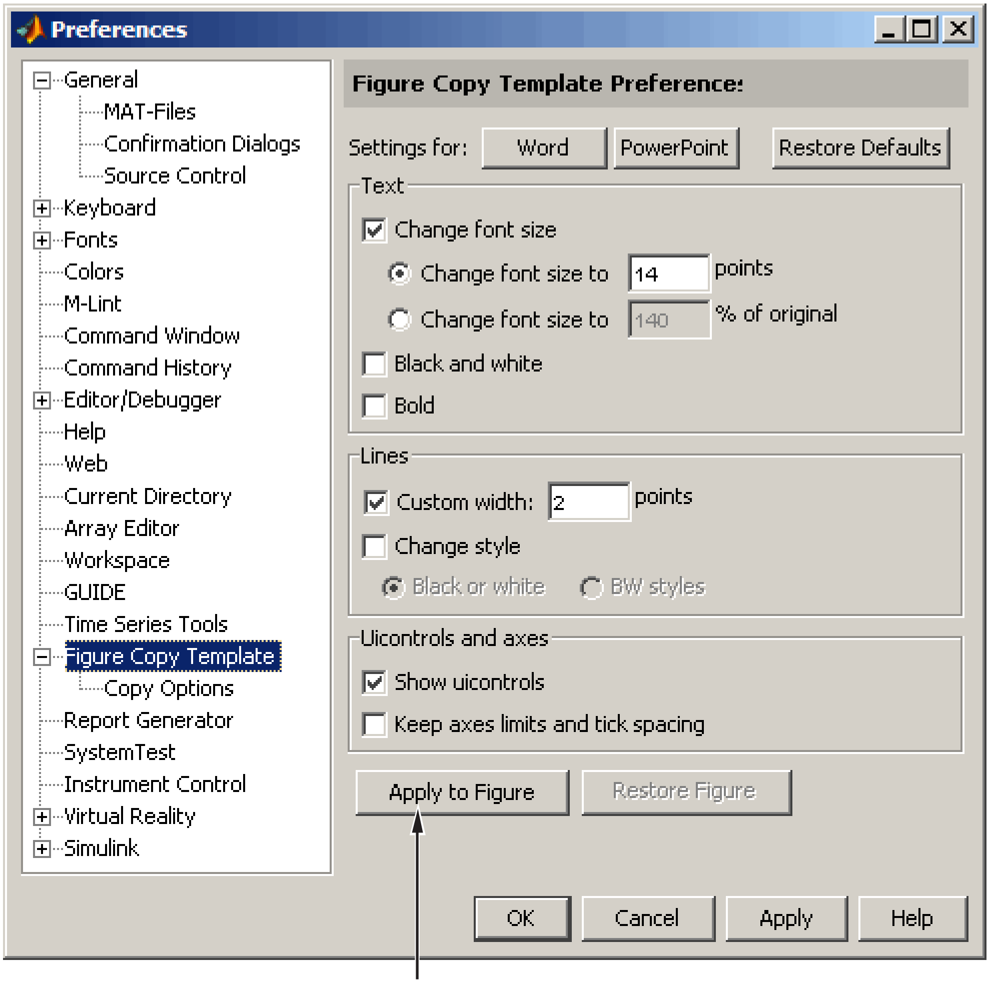
If you transfer your figures to Microsoft Word or PowerPoint applications, you can set line and text characteristics to values recommended for those applications. The Figure Copy Template Preferences dialog box provides Word and PowerPoint options to make these settings, or you can set certain line and text characteristics individually.

You can change line and text characteristics:

- "Using the Graphical User Interface" on page 7-64
- "Using MATLAB Commands" on page 7-65

Using the Graphical User Interface

To open Figure Copy Template Preferences, select **Preferences** from the **File** menu, and then click **Figure Copy Template** in the left pane.



Note the difference between **Apply to Figure** and **Apply**.
Use **Apply to Figure** to modify the figure in the figure window.
Use **Apply** or **OK** to save your preferences.

Settings you can change in the Figure Copy Template Preferences dialog box are as follows:

Microsoft Word or PowerPoint. Click **Word** or **PowerPoint** to apply recommended MATLAB settings.

Text. Use options under **Text** to modify the appearance of all text in the figure. You can change the font size, change the text color to black and white, and change the font style to bold.

Lines. Use the **Lines** options to modify the appearance of all lines in the figure:

- **Custom width** — Change the line width.
- **Change style (Black or white)** — Change colored lines to black or white.
- **Change style (B&W styles)** — Change solid lines to different line styles (e.g., solid, dashed, etc.), and black or white color.

UIControls and axes. If your figure includes user interface controls, you can choose to show or hide them by clicking **Show uicontrols**. Also, to keep axes limits and tick marks as they appear on the screen, click **Keep axes limits and tick spacing**. To allow autoamtic scaling of axes limits and tick marks based on the size of the printed figure, clear this box.

Note Changes you make using Print Preview affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

You can use the `set` function on selected graphics objects in your figure to change individual line and text characteristics.

For example, to change line width to `1.8` and line style to a dashed line, use

```
lineobj = findobj('type', 'line');
set(lineobj, 'linewidth', 1.8);
set(lineobj, 'linestyle', '--');
```

To change the font size to 15 points and font weight to bold, use

```
textobj = findobj('type', 'text');
set(textobj, 'fontunits', 'points');
set(textobj, 'fontsize', 15);
set(textobj, 'fontweight', 'bold');
```

Setting the Line and Text Color

When colored lines and text are dithered to gray by a black-and-white printer, it does not produce good results for thin lines and the thin lines that make up text characters. You can, however, force all line and text objects in the figure to print in black and white, thus improving their appearance in the printed copy. When you select this setting, the lines and text are printed all black or all white, depending on the background color.

The default is to leave lines and text in the color that appears on the screen.

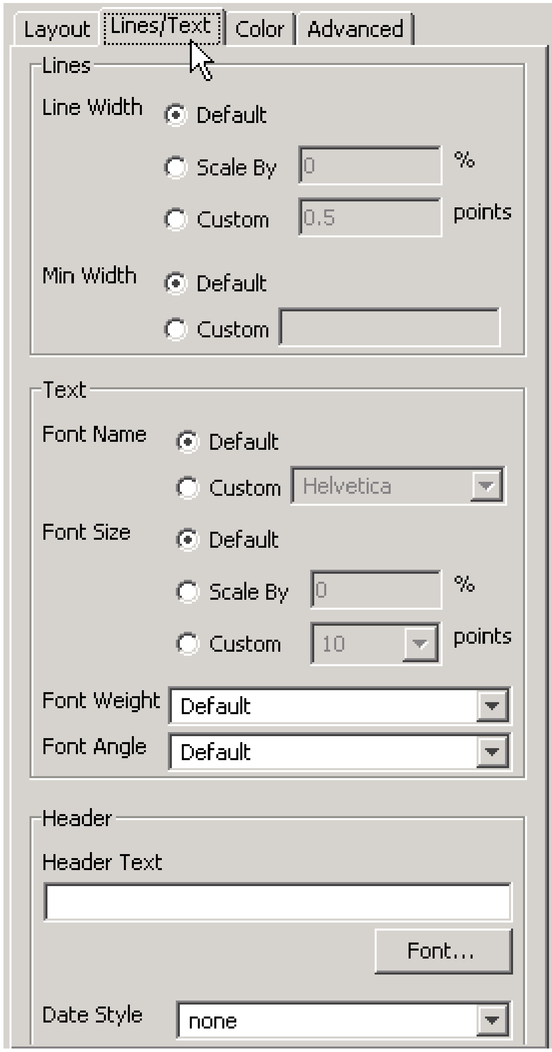
Note Your background color might not be the same as what you see on the screen. See the **Color** tab for an option that preserves the background color when printing.

You can change the resolution used to print a figure:

- "Using the Graphical User Interface" on page 7-66
- "Using MATLAB Commands" on page 7-68

Using the Graphical User Interface

Select **Print Preview** from the figure window's **File** menu to open the Print Preview dialog box. Select the **Lines and Text** tab to make changes to the color of all lines and text on the printed page. The controls for the **Lines and Text** tab are shown below:



Settings you can change in **Lines and Text** are as follows:

Lines. The default option in this panel causes lines to print at the same width they are portrayed in the figure window. You can scale line width from 0 percent upwards for printing using the **Scale By** field. To print lines at a particular point size, select **Custom**. All lines on the plot will be the same weight when you use the **Custom** option; the **Scale By** option respects relative line weight.

When you scale lines downward, you can prevent them from becoming too faint by setting the **Min Width** option to **Custom** and specifying a minimum line width in points in that field.

Text. The default is to print text in the same font and at the same size as it is in the figure. To change the font (for all text) select **Custom** and choose a new font from the drop-down list that is then enabled. Scale the font size using the **Scale By** option. To print text at a particular point size, select **Custom**. All text on the plot will be printed at the point size you specify when you use the **Custom** option; the **Scale By** option respects relative font size. You can specify the **Font Weight** (normal, light, demi, or bold) and **Font Angle** (`normal`, `italic`, or `oblique`) for all text as well, using the drop-down menus at the bottom of the **Text** panel.

Header. Type any text that you want to appear at the top of the printed figure in the **Header Text** edit field. If you want today's date and/or time appended to the header text, select the appropriate format from the **Date Style** popup menu. To specify and style the header font (which is independent of the font used in the figure), click the **Font** button and choose a font name, size, and style from the **Font** selection window that appears.

Note Changes you make using Print Preview affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

There is no equivalent MATLAB command that sets line and text color depending on background color. Set the color of lines and text using the `set` function on either line or text objects in your figure.

This example sets all lines and text to black:

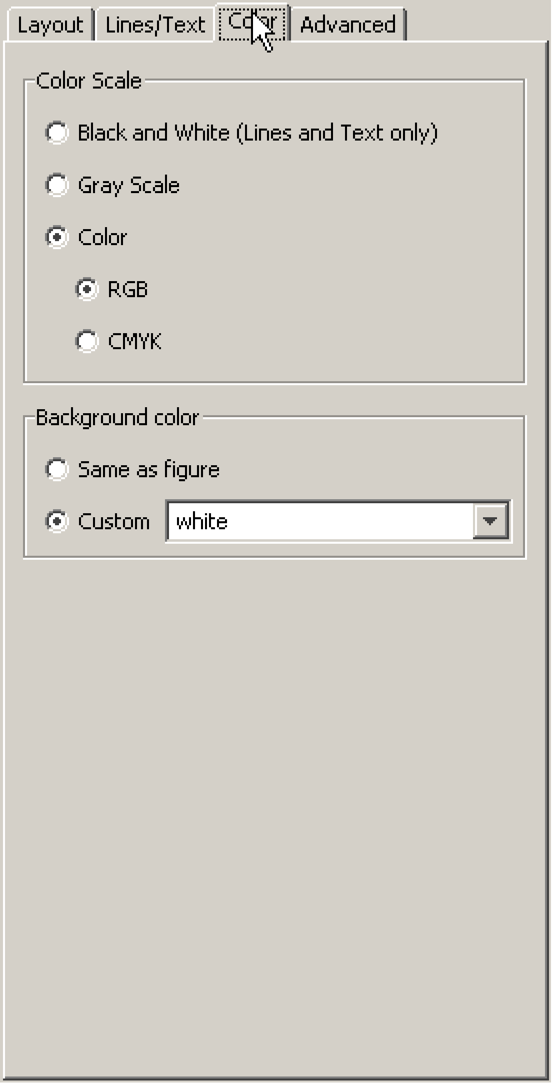
```
set(findobj('type', 'line'), 'color', 'black');
set(findobj('type', 'text'), 'color', 'black');
```

Specifying a Colorspace for Printing and Exporting

By default, color output is in the RGB color space (red, green, blue). If you plan to publish and print MATLAB figures using printing industry standard four-color separation, you might want to use the CMYK color space (cyan, magenta, yellow, black).

Using the Windows Graphical User Interface

Select **Print Preview** from the figure window's **File** menu to open the Print Preview dialog box. Select the **Color** tab to make changes to the color of all lines and text on the printed page. The controls for the **Color** tab are shown below:



You can print the contents of your figure in color, grayscale, or black-and-white by selecting the appropriate button in the panel. When you select **Color**, you can choose between an **RGB** (red/green/blue) or a **CMYK** (cyan/magenta/yellow/black) color specification, if your printer is capable of it.

Independently of the **Color Scale** controls, you can specify a **Background color** for printing. Select **Same as figure** to use the color used in the figure itself (default is gray), or specify a **Custom** color from the combo box popup menu. The choices are `black`, `white`, and several RGB color triplet values; you type any valid MATLAB colorspec in this field as well, such as `g`, `magenta`, or `.3 .4 .5`.

The background color you specify is respected even if you choose **Black and White** or **Gray Scale** in the **Color Scale** panel.

Using the Graphical User Interface on UNIX Platforms

1. Select **Print** from the figure window's **File** menu.
2. Click the **Appearance** tab.
3. In the **Color Appearance** panel, select **Color**.
4. Click **Print**.

On any platform, you can also indicate whether to print in color, grayscale or black-and-white with the Print Preview dialog box.

Using MATLAB Commands

Use the `-cmyk` option with the `print` function. This example prints the current figure in CMYK using a PostScript Level II color printer driver.

```
print -dpsc2 -cmyk
```

Excluding User Interface Controls form Printed Output

User interface controls are objects that you create and add to a figure. For example, you can add a button to a figure that, when clicked, conveniently runs another M-file. By default, user interface controls are included in your printed or exported figure. This section shows how to exclude them.

Using the Graphical User Interface

1. Open the Print Preview dialog box by selecting **Print Preview** from the figure window's **File** menu, and then select the **Advanced** tab.

2. Under **Miscellaneous**, clear the **Print UIControls** check box.

3. Click **Close**.

Using MATLAB Commands

Use the `-noui` switch. This example specifies a color PostScript driver and excludes UI controls.

```
print -dpsc -noui
```

This example exports the current figure to a color EPS file and excludes UI controls.

```
print -depsc -noui myfile.eps
```

Producing Uncropped Figures

In most cases, MATLAB crops the background tightly around the objects in the figure. Depending on the printer driver or file format you use, you might be able to produce uncropped output. An uncropped figure has increased background area and is often desirable for figures that contain UI controls.

The setting you make changes the PostScript `BoundingBox` property saved with the figure.

Using MATLAB Commands

Use the `-loose` option with the `print` function. On Windows platforms, the uncropped option is only available if you print to a file.

This example exports the current figure, uncropped, to an EPS file.

```
print -deps -loose myfile.eps
```

Choosing a Graphics Format

In this section...

What Are Graphic Formats?

A graphics file format is a specification for storing and organizing data in a file. MATLAB support exists for many different graphics file formats. Some are built-in and others are Ghostscript formats. File formats also differ in color support, graphics style (bitmap or vector), and bit depth.

This section provides information to help you decide which graphics format to use when exporting your figure to a file or to the Windows clipboard. It covers

Before deciding on a graphics format, check what formats are supported by your target application and platform. See the `print` reference page for a complete list of supported MATLAB graphics formats. Once you decide on which format to use in exporting your figure, follow the instructions in "Exporting to a File" on page 7-20 or "Exporting to the Windows or Macintosh Clipboard" on page 7-32.

Frequently Used Graphics Formats

Here are some of the more frequently used graphics formats. For a complete list, see the Graphics Format table on the `print` reference page. For a more complete description of these formats, see "Description of Selected Graphics Formats" on page 7-80.

Format	Description	Command Line -device Parameter
EPS color, and black and white	Export line plots or simple graphs to a file. **Note** An EPS file does not display within some applications unless you add a TIFF preview image to it. See the example "Exporting in EPS Format with a TIFF Preview" on page 7-40.	`-deps` (black and white) `-depsc` (color) `-depsc -tiff` (TIFF preview)
JPEG 24-bit	Export plots with surface lighting or transparency to a file. This format can be displayed by most Web browsers.	`-djpeg` `-djpeg`*`number`*, where *`number`* is the compression.
TIFF 24-bit bitmap color	Export plots with surface lighting or transparency to a file. Widely available. A good format to choose if you are not sure what formats your application supports.	`-dtiff`
BMP 8-bit color bitmap	Export a figure to the clipboard (Windows only).	`-dbitmap`
EMF color vector format	Export a figure to the clipboard (Windows only).	`-dmeta`

Factors to Consider in Choosing a Format

There are at least five main factors to consider when choosing a graphics format to use in exporting a figure:

- Implementation — Is it a built-in MATLAB or Ghostscript format?
- Graphics Format — Is it bitmap or vector graphics format?
- Bit Depth — What bit depth does the format offer?
- Color Support — What color support does it have?
- Model/Publication — Is it a Simulink® model or specific publication type?

The Graphics Format table on the `print` reference page provides information on the first four of these factors for each format that MATLAB supports.

Built-In MATLAB or Ghostscript Formats

Some graphics formats are built-in MATLAB formats and others are provided by Ghostscript. In some cases (such as the Windows Bitmap format), the format is available both as a built-in format and a Ghostscript format. In general, when this is the case, we recommend that you choose the MATLAB format, especially if you plan to read the image back into Windows later.

The choice of MATLAB versus Ghostscript formats is important when any of these properties affects your output:

- “Font Support” on page 7-77
- “Resolution” on page 7-77
- “Importing into the MATLAB Workspace” on page 7-78

Choosing Bitmap or Vector Graphic Output

Windows file formats are created using either bitmap or vector graphics. Bitmap formats store graphics as 2-D arrays of pixels. Vector formats use drawing commands to store graphics as geometric objects. Whether to use a bitmap or vector format depends mostly on the type of objects in your figure.

The choice of bitmap versus vector graphics is important when any of these properties or capabilities affects your output:

- “Degree of Complexity” on page 7-78
- “Lighting and Transparency” on page 7-78

- Line and text quality
- "File Size" on page 7-79
- "Resizing After Import" on page 7-79

To create vector output, the Painters renderer is required. Under some circumstances you might need to manually select it in the Print Preview or Export Setup GUI. The painters renderer does not support lighting or transparency.

To create bitmap output, either the OpenGL or the Z-buffer renderer is required. Under some circumstances you might need to manually select one of these in the Print Preview or Export Setup GUI. These renderers both support lighting, but only OpenGL supports transparency.

See "Impact of Rendering Method on the Output" on page 7-79 for more information.

Bit Depth

Bit depth is the number of bits a format uses to store each pixel. This determines the number of colors the exported figure can contain.

Bit depth applies mostly to bitmap graphics. An 8-bit image uses 8 bits per pixel (bpp), enabling it to define 2^8, or 256, unique colors. The other supported bit depths are 1-bit (2 colors), 4-bit (16 colors), and 24-bit (16 million colors).

In vector files that don't normally have a bit depth, the color of objects is specified by drawing commands stored in the file. However, vector files can contain bitmaps under the following conditions:

- Image objects saved in vector formats are always saved as bitmaps, regardless of the rendering method used.
- For vector files created using the OpenGL or Z-buffer renderer, everything in the figure is saved as a bitmap.

The Graphics Format table on the `print` reference page indicates the bit depth of each format. If file size is not critical, make sure you choose a format with a bit depth that supports the number of colors or shades of gray in your displayed figure.

Color Support

Each graphics format can produce color, grayscale, or monochrome output. Check the Graphics Format table to see the level of color support for each format type. To preserve the color in your exported file, you must select a color graphics format. Bit depth also affects color.

Exporting Simulink Models

Simulink models can only be exported to EPS or a Ghostscript format. You can only use the `print` function to export a model, not the Export dialog box.

High Resolution or Web Publications

If you want to use a figure in a journal or other publication, use a format that enables you to set a high resolution, such as TIFF or EPS.

If you want to use a figure in a Web publication, use either the PNG or the JPEG format. If you need to save an image as a GIF file, you can use the `imwrite` function. You need to convert M-by-N-by-3 truecolor CData (such as the `getframe` function provides) to an M–by–N 8–bit array and a colormap in order to write a GIF. Alternatively, you can export your figure as a TIFF file and convert it to a GIF using another software application, or capture a figure as an image using a screen capture utility and save it in formats the utility supports.

Properties Affected by Choice of Format

The figure properties listed in this section are affected when you select a graphics format when exporting to a file or the Windows clipboard.

Font Support

Ghostscript formats support a limited number of fonts. If you use an unsupported font, `Courier` is substituted. See "PostScript and Ghostscript Supported Fonts" on page 7-88 for more information.

Resolution

Generally, higher resolution means higher quality. Your choice of resolution should be based in part on the device to which you will ultimately print it. Experimentation with different resolution settings can be helpful.

You cannot change the resolution of a Ghostscript format. The resolution is low (72 dpi) and might not be appropriate for publications.

Importing into the MATLAB Workspace

If you want to read an exported figure back into the MATLAB environment, it is best to use one of the built-in MATLAB formats. You should not use PostScript or a proprietary format such as Adobe Illustrator (`.ai`), Windows metafile (`.emf`), or portable document format (`.pdf`) files.

Degree of Complexity

Bitmaps are preferable for high-complexity plots, where complexity is determined by the number of polygons, the number of polygons with interpolated shading, the number of markers, the presence of truecolor images, and other factors. An example of a high-complexity plot is a surface plot that uses interpolated shading.

Vector formats are preferable for most 2-D plots and for some low-complexity surface plots.

Lighting and Transparency

Surface lighting and transparency are only supported by bitmap graphics formats. If you use a vector format, the lighting and transparency disappear. Of the two renderers intended for bitmaps (OpenGL and Z-buffer) only OpenGL supports transparency.

Note If you export to an EPS (vector) file using the Painters renderer and include a TIFF preview, the preview image is a bitmap and shows lighting or transparency when displayed on your screen. Remember that the underlying format vector file, which is what normally gets printed, does not support these features.

Lines and Text

Generally, vector formats create better lines and text than bitmap formats. MATLAB renderers do not antialias lines or text.

File Size

In general, bitmap formats produce smaller files for complex plots than vector formats, and vector formats produce smaller files for simple plots than bitmap formats.

You can calculate the size of a figure exported to an uncompressed bitmap by multiplying the figure size by its resolution and the bit depth of the chosen format. For example, if a figure is 2 inches by 3 inches and has a resolution of 100 dpi (dots per inch), it will consist of (2x100)x(3x100), or 60,000 pixels. If exported to an 8-bit file, it uses 480,000 bits, or 60 KB. If exported to a 24-bit file, it uses three times the number of bytes, or 180 KB.

Vector format file size is affected by the complexity and number of objects in your figure. As the complexity and number of objects increase, the number of drawing commands increases.

Resizing After Import

You can resize a vector graphics figure after importing it into another software application without losing quality. (Not all applications that support vector formats enable you to resize them.)

This is not true of bitmap formats. Resizing a bitmap causes round-off errors that result in jagged edges and degradation of picture quality. This degradation is particularly obvious in lines and text and is highly discouraged.

Color

The Graphics Format table on the `print` reference page indicates the color support and bit depth of each format. If file size is not critical, make sure you choose a format with a bit depth that supports the number of colors or shades of gray in your displayed figure.

Impact of Rendering Method on the Output

If you specify a bitmap format when exporting, the exported file always contains a bitmap regardless of your current renderer setting. If you have the renderer set to Painters, which normally produces a vector format, that setting is ignored under these circumstances.

Vector format files, however, can store your figure as a vector or bitmap graphic depending on the renderer used to export it. If you do not specify a rendering method and OpenGL or Z-buffer is chosen automatically, your exported vector file contains a bitmap. If you want your figure exported as a vector graphic, be sure to set the rendering method to Painter's.

Description of Selected Graphics Formats

This section contains details about some of the export file formats MATLAB supports. For information about formats not listed here, consult a graphics file format reference.

Formats covered in this section are

- "Adobe Illustrator 88 Files" on page 7-80
- "EMF Files" on page 7-81
- "EPS Files" on page 7-81
- "TIFF Files" on page 7-82
- "JPEG Files" on page 7-83

Adobe Illustrator 88 Files

Adobe Illustrator (ILL) is a vector format that is fully compatible with Adobe Illustrator software. An Illustrator file created in MATLAB can be further processed with Adobe Illustrator running on any platform. (When you view it in Illustrator, it has no template.)

By default, Illustrator files are color and saved in portrait orientation. The Illustrator `group` command is used to give the illustrations a hierarchy similar to that of the Handle Graphics or Simulink graphic.

Some limitations of the Illustrator format are

- Interpolated patches and surfaces cannot be created. The color of each polygon is determined by the average of the `CData` values for all of the polygon's vertices.
- Images cannot be exported in this format.

- The resolution setting of 72 dpi cannot be changed.
- No fonts are downloaded to the Illustrator file. Any unavailable fonts are replaced with fonts that are available.

EMF Files

Enhanced Metafiles (EMF) are vector files similar in nature to Encapsulated PostScript (EPS), capable of producing near publication-quality graphics. EMF is an excellent format to use if you plan to import your image into a Microsoft application and want the flexibility to edit and resize your image once it has been imported. It is the only supported MATLAB vector format you can edit from within a Microsoft application. (Your editing ability is limited. For the best results, do all your editing in Microsoft.)

A drawback of using EMF files is that they are generally only supported by Windows based applications.

EPS Files

The Encapsulated PostScript (EPS) vector format is the most reliable and consistent file format that MATLAB printing and export supports. It is widely recognized in desktop publishing and word processing packages on both UNIX and Windows platforms. EPS is the only MATLAB supported export format that can produce CMYK output. (PostScript printer drivers also support this feature.)

This format is your best choice for producing publication-quality graphics. It might not be appropriate for figures containing interpolated shading because it creates a very large file that is difficult to print. For such figures, use the TIFF format with a high-resolution setting. For more information about format choices, see "Choosing Bitmap or Vector Graphic Output" on page 7-75.

When imported into Microsoft applications, an EPS file does not display unless you add a TIFF preview image to it.

The preview image is simple to add (see the next section, "Creating a Preview Image"). However, if you print your file to a non-PostScript printer, the TIFF preview is used as the printed image. The resolution of the preview image is 72 dpi, resulting in much lower quality than the EPS image. If there is no preview image, your printout to a non-PostScript printer contains an error

message in place of the graphic. Many high-end graphics packages, like Adobe Illustrator, can print an EPS file to a non-PostScript printer.

You cannot edit figures when using EPS files in Microsoft applications; they can only be annotated.

Note The best vector format to use with Microsoft applications is EMF. See "EMF Files" on page 7-81.

EPS format has limited font support. When you export a graphic to the EPS file format, ino attempt is made to determine whether the fonts you have used in your axes text objects are supported by the EPS format. Unsupported fonts are replaced with `Courier`.

Creating a Preview Image. You cannot create TIFF preview images using the graphical user interface. Use the `print` command with the `-tiff` switch. For example, to create an EPS Level 2 image with TIFF preview in file `myfile.eps`, type

```
print -depsc2 -tiff myfile.eps
```

TIFF Files

The Tagged Image File Format (TIFF) is a very widely used bitmap format and can produce publication-quality graphics if you use a high-resolution setting (such as 200 or 300 dpi).

TIFF is a good format to choose if you are not sure what formats your target application supports, or if you want to import the graphic into more than one application without having to export it to several different formats. It can also be imported into most image-processing applications and converted to other formats, if necessary. For example, the `print` command does not produce GIF files, but there are many applications that can convert TIFF files to GIF. You can also use `getframe` to create a snapshot of a figure and `imwrite` to save that image as a GIF file.

JPEG Files

The Joint Photographic Experts Group (JPEG) bitmap format is one of the dominant formats used in Web graphics. The 24-bit version that MATLAB supports carries more color information than the popular GIF format.

JPEG files always use JPEG compression. This is a lossy compression scheme, meaning that some data is thrown away during compression. When you export to a JPEG image, you can set the amount of compression to use. The more compression you use, the more data is thrown away. The compression amount is referred to as JPEG quality, where the highest setting results in the highest quality image, but the lowest amount of compression.

Setting JPEG Quality. You cannot set the quality using the graphical user interface. Use the `print` command with the `-djpeg` format switch, including the desired quality value as a suffix. This example exports to a JPEG file using a quality setting of 100.

```
print -djpeg100 myfile.jpg
```

By default, a quality setting of 75 is used. Possible values are from 1 to 100. The highest setting of 100 still results in some data loss, although the result is usually visually indistinguishable from the original.

How to Specify a Format for Exporting

To select a graphics format to use when exporting, choose a format from the Graphics Format table on the `print` reference page, and specify that format in either the Export dialog box or in the MATLAB `print` function.

Using the Graphical User Interface

When exporting your figure to a file:

1. Select **Export** from the figure window's **File** menu.
2. Select a format from the **Save as type** list box.
3. Enter the filename you want to use and browse for the directory to save the file in.
4. Click **Save**.

Using MATLAB Commands

To specify a nondefault graphics format for the figure you are exporting, include the `-d` switch with the `print` command. For example, to export the current figure to file `spline2d.eps`, with 600 dpi resolution, and using the EPS color graphics format, type

```
print -r600 -depsc spline2d
```

Note When printing, the `print -d` option specifies a printer driver. When exporting, the `print -d` option specifies a graphics format.

Choosing a Printer Driver

In this section...

What Are Printer Drivers?

A printer driver formats your figure into instructions that your printer understands. There are two main types of MATLAB printer drivers: built-in and Ghostscript. See the Printer Driver table on the `print` reference page for a complete list of supported drivers. Specifying the printer driver does not change the selected printer. This following sections provide information to help you decide which printer driver to use when printing your figure.

Built-in MATLAB Drivers

Built-in MATLAB drivers are written specifically for it and include Windows, PostScript, and HPGL output formats.

The built-in Windows printer drivers enable your print requests to work with the Windows Print Manager. The Print Manager enables you to monitor printer queues and control various aspects of the printing process.

HPGL support is provided for the HP 7475A plotter and fully compatible plotters. HPGL files can also be imported into documents of other applications, such as Microsoft Word, although add-on filters for them may be needed.

Ghostscript Drivers

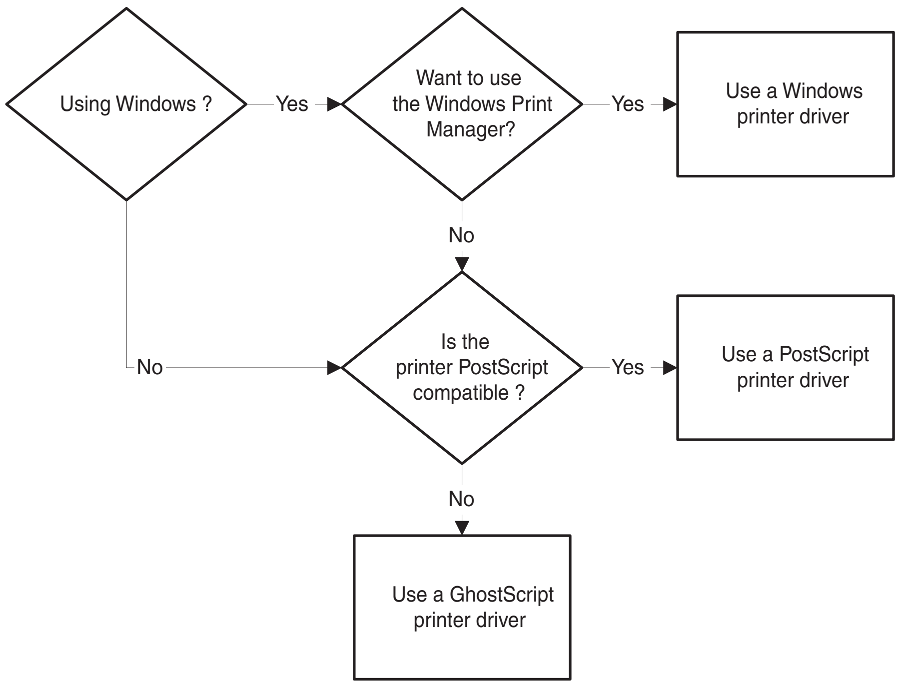
Ghostscript drivers use Ghostscript to convert your figure into printer-model-specific instructions. MATLAB generates a PostScript representation of the figure and Ghostscript generates the printer instructions from that. Examples of Ghostscript drivers are Epson and HP.

Factors to Consider in Choosing a Driver

The choice of printer driver depends upon several considerations:

- What platform you are using
- What kind of printer you have
- What color model you want to use
- What font support you need
- Any driver-specific settings you need

The following flowchart gives an overview of how to choose a driver based on the platform you are using and the type of printer you have.



Deciding What Type of Printer Driver to Use

Platform Considerations

On Windows systems, you can use any of the driver types shown in the flowchart. If you use the Windows driver, you can use the Windows Print Manager.

On UNIX, you can use either PostScript or Ghostscript drivers.

On either platform, if you have a PostScript-compatible printer, it is better to use a PostScript driver than a Ghostscript driver, because doing so avoids the unnecessary Ghostscript conversion step and is likely to create more accurate renditions.

Printer Type

Printer support is different among the Windows, PostScript, and Ghostscript drivers. Consult the manual for your printer to see what driver to use.

Windows drivers support most printer models, but sometimes the printer's native driver is incompatible with the MATLAB Windows driver. If you are getting printing errors, see "Trouble with Windows Native Drivers" on page 7-90.

Some Ghostscript drivers are specific to certain printer models. For example, different drivers support the HP DeskJet 500, 500C, and 550C models, plus a generic driver for the series. When this is the case, try the model-specific driver first. If that doesn't work, try the generic driver.

Color Model

By default, a black-and-white driver is used. The built-in MATLAB and Ghostscript drivers print both color and black and white. The Printer Drivers table on the `print` reference page indicates which drivers are color.

Colored surfaces and images print in grayscale when you use a black-and-white driver. Colored lines and text can be printed in color, grayscale, or black and white, depending on the color support of the driver and color capability of your printer. Results can vary depending on whether images, text, lines, patches, or surfaces are being printed.

Font Support

In MATLAB, the fonts supported for printing depend upon the MATLAB printer driver you specify and sometimes upon which platform you are using.

PostScript and Ghostscript Supported Fonts. The table below lists the fonts supported by the MATLAB PostScript and Ghostscript drivers. This same set of fonts is supported on both Windows and UNIX. If you use a font that is not on this list, it is replaced with Courier.

AvantGarde	Helvetica-Narrow	Times-Roman
Bookman	NewCenturySchlbk	ZapfChancery
Courier	Palatino	ZapfDingbats
Helvetica	Symbol	

If you set the font using the `set` function, use the names exactly as shown above. This example sets the font of the current text object to Helvetica-Narrow using MATLAB commands.

```
set(gca, 'FontName', 'Helvetica-Narrow');
```

If you use the Property Editor dialog box (available under **Axes Properties** or **Current Object Properties** on the **Edit** menu) to set the font, the list of available fonts shows those that are supported by your system. If you choose one that is not in the table above, your resulting file uses `Courier`.

Windows Drivers Supported Fonts. The MATLAB Windows drivers support any system-supported font. To see the list of fonts installed on your system, open the **Font name** list on the **Text** or **Style** tab of the Property Editor.

If you use the `set` function to set fonts, type in the name exactly as it appears in the Property Editor. For example, if you have the Script font installed on your system, set the title of your figure to `Script` using the following code.

```
h = get(gca, 'Title');
set(h, 'FontName', 'Script');
```

If you specify a font supplied with MATLAB that is not available on your platform as a system font, figures might not print or export properly.

HPGL Driver Supported Fonts. HPGL drivers support only one font. However, you can set its size and color.

Settings That Are Driver Specific

Some print settings are only supported by specific drivers. This table summarizes the settings and which driver supports them.

Setting	Driver(s)
Appending figures to a PostScript file	PostScript
BoundingBox (setting figure to print uncropped)	PostScript, Ghostscript
CMYK	PostScript
Resolution set with user interface	PostScript, Windows
Resolution set with `print` function	PostScript

Driver-Specific Information

This section provides additional information about the various types of printer drivers available to MATLAB users. It covers the following topics:

- "Setting the Windows Driver" on page 7-90
- "Trouble with Windows Native Drivers" on page 7-90
- "Level 1 or Level 2 PostScript Drivers" on page 7-91
- "Early PostScript 1 Printers" on page 7-91
- "Background Fills in HPGL Drivers" on page 7-91
- "Color Selection in HPGL Drivers" on page 7-91
- "Limitations of HPGL Drivers" on page 7-92

Setting the Windows Driver

When you specify a Windows driver (-dwin or -dwinc), this is interpreted to mean that the print request will use the Windows Print Manager. It also means that the default Windows driver will be assigned based on your current printer's color property setting. In other words, MATLAB does not differentiate between -dwin or -dwinc in printopt.m and you might not get the expected output color: if you choose -dwin, lines and text will print in black and white; with -dwinc, lines and text print in their screen colors (assuming your printer does print in color).

There are two ways to ensure that -dwin or -dwinc are used: specify the driver when you print, or use the printer's Document Properties dialog box to set the default driver.

You can use the printer's Document Properties dialog box to set the default driver for all print requests. This dialog box sets the printer's color property, which in turn sets the default Windows driver.

To access this dialog box, click the **Properties** button on the Windows Print or Print Setup dialog box. See your Windows and printer's documentation if you need help with this dialog box. Document Properties dialog boxes vary from printer to printer.

Sometimes, even when you use the Windows Document Properties dialog box, you can receive incorrect color results because some Windows printers return inaccurate information about their color property setting.

Trouble with Windows Native Drivers

Occasionally, printing problems are due to a bug in the native printer driver or an incompatibility between the native printer driver and the MATLAB driver.

If you are having trouble, try installing a different native printer driver. A newer version might be available from the manufacturer or reseller. You may also be able to use the native driver from a different printer, such as an earlier model from the same manufacturer.

If this doesn't help, try using a PostScript or Ghostscript driver.

Level 1 or Level 2 PostScript Drivers

Choosing between the Level 1 and Level 2 MATLAB PostScript drivers does not affect the quality of your output. Make the choice based on what your printer supports and on any file size or speed concerns.

Level 1 PostScript produces good results on a Level 2 printer, but Level 2 PostScript does not print properly on a Level 1 printer.

Level 2 PostScript files are generally smaller and render more quickly than Level 1 files. If your printer supports Level 2 PostScript, use one of the Level 2 drivers. If your printer does not support Level 2, or if you're not sure, use a Level 1 driver.

Early PostScript 1 Printers

If you have an early PostScript 1 printer, such as some of the PostScript printers manufactured before 1990, you may notice problems in the text of PostScript printouts. Your printer might not support the `ISOLatin1Encoding` operator that the MATLAB driver uses for PostScript files. If this is the case, use the Adobe PowerPoint default character-set encoding. You can specify this by using the `-adobecset` option with the `print` command.

Background Fills in HPGL Drivers

The HPGL driver cannot do background fills. Therefore, you should ensure that your figure is set to print with a white background (the default), and that any lines and text in your figure are drawn in a color dark enough to be seen on a white background. For more information about background color, see "Setting the Background Color" on page 7-62.

Color Selection in HPGL Drivers

The HP 7475A plotter supports six pens, none of which can be white. If the MATLAB driver tries to draw in white while rendering in HPGL mode, the driver ignores all drawing commands until a different color is chosen.

Pen 1, which is assumed to be black, is used for drawing axes. The remaining pens are used for the first five colors specified in the `ColorOrder` property of the current axes object. If `ColorOrder` specifies fewer than five colors, the unspecified pens are not used.

For Simulink systems, which ordinarily use a maximum of eight colors, the six pens available on the plotter are assumed to be

- Pen 1: black
- Pen 2: red
- Pen 3: green
- Pen 4: blue
- Pen 5: cyan
- Pen 6: magenta

If you attempt to draw a MATLAB graphic object containing a color that is not a known pen color, the driver chooses the nearest approximation to the unlisted color.

Limitations of HPGL Drivers

The HPGL driver has these limitations:

- Display colors and plotted colors sometimes differ.
- Areas (faces on mesh and surface plots, patches, blocks, and arrowheads) are not filled.
- There is no hidden line or surface removal.
- Text is printed in the plotter's default font.
- Line width is determined by pen width.
- Images and UI controls cannot be plotted.
- Interpolated edge lines between two vertices are drawn with the pen whose color best matches the average color of the two vertices.
- Figures cannot be rendered using Z-buffer or OpenGL; this driver always uses the Painter's algorithm.

How to Specify the Printer Driver to Use

If you need to use a driver other than the default driver for your system, choose a new driver from the Printer Driver table on the `print` reference page, and set it either as a new default or just for the current figure you are working on.

Setting the Default Driver for All Figures

If you do not indicate a specific printer driver, MATLAB uses the default driver specified by the variable `dev` in the `printopt.m` file. The factory default driver depends on the platform.

Platform	Factory Default Printer Driver	Driver Code
Windows	Black-and-white Windows	`-dwin`
UNIX & Macintosh	Black-and-white Level II PostScript	`-dps2`

To change the default driver for all figures, edit `printopt.m` and change the value for `dev` to match one of the driver codes listed in the Printer Drivers table on the `print` reference page (`printopt.m` contains instructions for modifying it). See "Setting Defaults Across Sessions" on page 7-8 for details.

Setting a Driver for the Current Figure Only

You can change the printer driver from the MATLAB command line. To specify a nondefault printer driver for the figure you are printing, include the `-d` switch with the `print` command. For example, to print the current figure using the MATLAB built-in Windows color printer driver `winc`, type

```
print -dwinc
```

Note When printing, the `print -d` option specifies a printer driver. When exporting, the `print -d` option specifies a graphics format.

Troubleshooting

In this section...

Introduction

This section describes some common problems you might encounter when printing or exporting your figure. If you don't find your problem listed here, try searching the Knowledge Base maintained by the MathWorks™ Technical Support Department. Go to `http://www.mathworks.com/support` and enter a topic in the search field.

Common Problems

Printing Problems

Printer Drivers

I am using a Windows printer driver and encountering problems such as segmentation violations, general protection faults, application errors, and unexpected output.

Try one of the following solutions:

- Check the table of `drivers` in the `print` reference page to see if there are other drivers you can try.
  - If your printer is PostScript-compatible, try printing with one of the MATLAB built-in PostScript drivers.
  - If your printer is not PostScript-compatible, see if one of the MATLAB built-in Ghostscript devices is appropriate for your printer model. These devices use Ghostscript to convert PostScript files into other formats, such as HP LaserJet and Canon BubbleJet.
- Contact the printer vendor to obtain a different native printer driver. The behavior you are experiencing might occur only with certain versions of the native printer driver. If this doesn't help and you are on a Windows

system, try reinstalling the drivers that were shipped with your Windows installation disk.

- Export the figure to a graphics-format file, and then import it into another application before printing it. For information about exporting MATLAB figures, see “Exporting to a File” on page 7-20.

PostScript Output

When I use the print function with the `-deps` switch, I receive this error message.

```
Encapsulated PostScript files cannot be sent to the printer.
File saved to disk under name 'figure2.eps'
```

As the error message indicates, your figure was saved to a file. EPS is a graphics file format and cannot be sent to a printer using a printer driver. To send your figure directly to a printer, try using one of the PostScript driver switches. See the table of drivers in the `print` reference page. To print an EPS file, you must first import it into a word processor or other software program.

Default Settings

My printer uses a different default paper type than the MATLAB default type of letter. How can I change the default paper type so that I don’t have to set it for each new figure?

You can set the default value for any property by adding a line to `startup.m`. Adding the following line sets the default paper type to A4.

```
set(0, 'DefaultFigurePaperType', 'A4');
```

In your call to `set`, combine the word `Default` with the name of the object `Figure` and the property name `PaperType`.

I set the paper orientation to landscape, but each time I go to print a new figure, the orientation setting is portrait again. How can I change the default orientation so that I won’t have to set it for each new figure?

See the explanation for the previous question. Adding the following line to `startup.m` sets the default paper orientation to landscape.

```
set(0, 'DefaultFigurePaperOrient', 'landscape')
```

Color vs. Black and White

I want the lines in my figure to print in black, but they keep printing in color.

You must be using a color printer driver. You can specify a black-and-white driver using the `print` function or the Print Preview dialog box to force the lines for the current figure to print in black. See “Setting the Line and Text Color” on page 7-66 for instructions.

A white line in my figure keeps coming out black when I print it.

There are two things that can cause this to happen. Most likely, the line is positioned over a dark background. The MATLAB default is to invert your background to white when you print, and changes any white lines over the background to black. To avoid this, retain your background color when you print. See “Setting the Background Color” on page 7-62.

The other possibility is that you are using a Windows printer driver and the printer is sending inaccurate color information to MATLAB.

I am using a color printer, but my figure keeps printing in black and white.

By default, MATLAB uses a black-and-white printer driver. You need to specify a color printer driver. For instructions, see “Choosing a Printer Driver” on page 7-85. If you are already using a Windows color driver, the printer might be returning inaccurate information about its color property. See “Driver-Specific Information” on page 7-89.

Printer Selection

I have more than one printer connected to my system. How do I specify which one to print my figure with?

You can use either the Print dialog box, or the MATLAB `print` function, specifying the printer with the `-P` switch. For instructions using either method, see "Selecting the Printer" on page 7-46.

Rotated Text

I have some rotated text in my figure. It looks fine on the screen, but when I print it, the resolution is poor.

You are probably using bitmapped fonts, which don't rotate well. Try using TrueType® fonts instead.

ResizeFcn Warning

I get a warning about my `ResizeFcn` being used when I print my figure.

By default, MATLAB resizes your figure when converting it to printer coordinates. That means it calls any `ResizeFcn` you have created for the figure and issues a warning. You can avoid this warning by setting the figure to print at screen size.

Exporting Problems

Background Color

I generated a figure with a black background and selected "Use figure color" from the Copy Options panel of the Preferences dialog box. But when I exported my figure, its background was changed to white.

You must have exported your figure to a file. The settings in **Copy Options** only apply to figures copied to the clipboard.

There are two ways to retain the displayed background color: use the Print Preview dialog box or set the `InvertHardCopy` property to `off`. See "Setting the Background Color" on page 7-62 for instructions on either method.

Default Settings

I want to export all of my figures using the same size. Is there some way to do this so that I don't have to set the size for each individual figure?

You can set the default value for any property by adding a line to `startup.m`. Adding the following line sets the default figure size to 4-by-3 inches.

```
set(0, 'DefaultFigurePaperPosition', [0 0 4 3]);
```

In your call to `set`, combine the word `Default` with the name of the object `Figure` and the property name `PaperPosition`.

I use the clipboard to export my figures as metafiles. Is there some way to force all of my copy operations to use the metafile format?

On Windows systems, use the **Copy Options** panel of the Preferences dialog box. Any settings made here, including whether your figure is copied as a metafile or bitmap, apply to all copy operations. See "Exporting to the Windows or Macintosh Clipboard" on page 7-32 for instructions.

Microsoft Word

I exported my figure to an EPS file, and then tried to import it into my Word document. My printout has an empty frame with an error message saying that my EPS picture was not saved with a preview and will only print to a PostScript printer. How do I include a TIFF preview?

Use the `print` command with the `-tiff` switch. For example,

```
print -deps -tiff filename
```

If you print to a non-PostScript printer with Word, the preview image is used for printing. This is a low-resolution image that lacks the quality of an EPS graphic. For more information about preview images and other aspects of EPS files, see "EPS Files" on page 7-81.

When I try to resize my figure in Word, its quality suffers.

You must have used a bitmap format. Bitmap files generally do not resize well. If you are going to export using a bitmap format, try to set the figure's size while it's still in MATLAB. See "Setting the Figure Size and Position" on page 7-47 for instructions.

As an alternative, you can use one of the vector formats, EMF or EPS. Figures exported in these formats can be resized in Word without affecting quality.

I exported my figure as an EMF to the clipboard. When I paste it into Word, some of the labels are printed incorrectly.

This problem occurs with some Microsoft Word and Windows versions. Try editing the labels in Word.

File Format

I tried to import my exported figure into a word processing document, but I got an error saying the file format is unrecognized.

There are two likely causes: you used the `print` function and forgot to specify the export format, or your word processing program does not support the export format. Include a format switch when you use the `print` function; simply including the file extension is not sufficient. For instructions, see "Exporting to a File" on page 7-20.

If this does not solve your problem, check what formats the word processor supports.

I tried to append a figure to an EPS file, and received an error message

You cannot append figures to an EPS file. The `-append` option is only valid for PostScript files, which should not be confused with EPS files. PostScript is a printer driver; EPS is a graphics file format.

Of the supported export formats, only HDF supports storing multiple figures, but you must use the `imwrite` function to append them. For an example, see the reference page for `imwrite`.

Size of Exported File

I've always used the EPS format to export my figures, but recently it started to generate huge files. Some of my files are now several megabytes!

Your graphics have probably become complicated enough that MATLAB is using the OpenGL or Z-buffer renderer instead of the Painter's renderer. It does this to improve display time or to handle attributes that Painter's cannot, such as lighting. However, using OpenGL or Z-buffer causes a bitmap to be stored in your EPS file, which with large figures leads to a large file.

There are two ways to fix the problem. You can specify the Painter's renderer when you export to EPS, or you can use a bitmap format, such as TIFF. The best renderer and type of format to use depend upon the figure. See "Choosing Bitmap or Vector Graphic Output" on page 7-75 if you need help deciding. For information about the rendering methods and how to set them, see "Selecting a Renderer" on page 7-54.

Making Movies

I am using MATLAB functions to process a large number of frames. I would like these frames to be saved as individual files for later conversion into a movie. How can I do this?

Use `getframe` to capture the frames, `imwrite` to write them to a file, and `movie` to create a movie from the files. For more information about using `getframe` and `imwrite` to capture and write the frames, see "Exporting with getframe" on page 7-29. For more information about creating a movie from the captured frames, see the reference page for `movie`.

You can also save multiple figures to an AVI file. AVI files can be used for animated sequences that do not need MATLAB to run. However, they do require an AVI viewer. For more information, see "Exporting Audio/Video Data" in the MATLAB Programming Fundamentals documentation.

Extended Operations

There are some export operations that cannot be performed using the Export dialog box.

You need to use the `print` function to do any of the following operations:

- Export to a supported file format not listed in the Export dialog box. The formats not available from the Export dialog box include HDF, some variations of BMP and PCX, and the raw data versions of PBM, PGM, and PPM.
- Specify a resolution.
- Specify one of the following options:
  - TIFF preview
  - Loose bounding box for EPS files
  - Compression quality for JPEG files
  - CMYK output on Windows platforms
- Perform batch exporting.

General Problems

Background Color

When I output my figure, its background is changed to white. How can I get it to have the displayed background color?

By default, when you print or export a figure, the background color inverts to white. There are two ways to retain the displayed background color: use the Print Preview dialog box or set the `InvertHardCopy` property to `off`. See "Setting the Background Color" on page 7-62 for instructions on either method.

If you are exporting your figure to the clipboard, you can also use the **Copy Options** panel of the Preferences dialog box. Setting the background here sets it for all figures copied to the clipboard.

Default Settings

I need to produce diagrams for publications. There is a list of requirements that I must meet for size of the figure, fonts types, etc. How can I do this easily and consistently?

You can set the default value for any property by adding a line to `startup.m`. As an example, the following line sets the default axes label font size to 12.

```
set(0, 'DefaultAxesFontSize', 12);
```

In your call to `set`, combine the word `Default` with the name of the object `Axes` and the property name `FontSize`.

Dimensions of Output

The dimensions of my output are huge. How can I make it smaller?

Check your settings for figure size and resolution, both of which affect the output dimensions of your figure.

The default figure size is 8-by-6 inches. You can use the Print Preview dialog box or the `PaperPosition` property to set the figure size. See “Setting the Figure Size and Position” on page 7-47.

The default resolution depends on the export format or printer driver used. For example, built-in MATLAB bitmap formats, like TIFF, have a default resolution of 150 dpi. You can change the resolution by using the `print` function and the `-r` switch. For default resolution values and instructions on how to change them, see “Setting the Resolution” on page 7-57.

I selected “Auto (actual size, centered)” from the Print Preview menu, but my output looks a little bigger, and my font looks different.

You probably output your figure using a higher resolution than your screen uses. Set your resolution to be the same as the screen’s.

As an alternative, if you are exporting your figure, see if your application enables you to select a resolution. If so, import the figure at the same resolution it was exported with. For more information about resolution and how to set it when exporting, see “Setting the Resolution” on page 7-57.

Axis and Tick Labels

When I resize my figure below a certain size, my x-axis label and the bottom half of the x-axis tick labels are missing from the output.

Your figure size might be too small to accommodate the labels. Labels are positioned a fixed distance from the *x*-axis. Since the *x*-axis itself is positioned a relative distance away from the window's edge, the label text might not fit. Try using a larger figure size or smaller fonts. For instructions on setting the size of your figure, see "Setting the Figure Size and Position" on page 7-47. For information about setting font size, see the Text Properties reference page.

In my output, the x-axis has fewer ticks than it did on the screen.

MATLAB has rescaled your ticks because the size of your output figure is different from its displayed size. There are two ways to prevent this: select **Keep screen limits and ticks** from the **Advanced** tab of the Print Preview dialog box, or set the `XTickMode`, `YTickMode`, and `ZTickMode` properties to `manual`. See "Setting the Axes Ticks and Limits" on page 7-60 for details.

UI Controls

My figure contains UI controls. How do I prevent them from appearing in my output?

Use the `print` function with the `-noui` switch. For details, see "Excluding User Interface Controls form Printed Output" on page 7-71.

Cropping

I can't output my figure using the uncropped setting (i.e., a loose BoundingBox).

Only PostScript printer drivers and the EPS export format support uncropped output. There is a workaround for Windows printer drivers, however. Using the `print` function, save your figure to a file that can be printed later. For an example see "Producing Uncropped Figures" on page 7-72.

Text Object Font

I have a problem with text objects when printing with a PostScript printer driver or exporting to EPS. The fonts are correct on the screen, but are changed in the output.

You have probably used a font that is not supported by EPS and PostScript. All unsupported fonts are converted to Courier. See "PostScript and Ghostscript Supported Fonts" on page 7-88 for the list of the supported fonts.

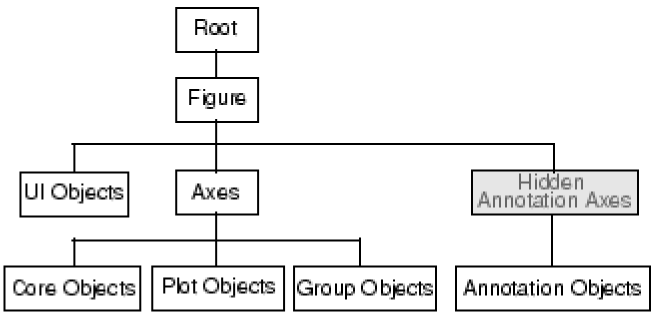
8

Handle Graphics Objects

Organization of Graphics Objects

Graphics objects are the basic drawing elements used by MATLAB software to display data. Each instance of an object is associated with a unique identifier called a *handle*. Using this handle, you can manipulate the characteristics (called object *properties*) of an existing graphics object. You can also specify values for properties when you create a graphics object.

These objects are organized into a hierarchy, as shown by the following diagram.



The hierarchical nature of the Handle Graphics technology is based on the interdependencies of the various graphics objects. For example, to draw a line object, MATLAB needs an axes object to orient and provide a frame of reference to the line. The axes, in turn, needs a figure window to display the axes and its child objects.

Types of Graphics Objects

In this section...

Introduction

There are two basic types of graphics objects:

- Core graphics objects — Used by high-level plotting functions and by composite objects to create plot objects.
- Composite objects — Composed of core graphics objects that have been grouped together to provide a more convenient interface.

Composite objects form the basis for four subcategories of graphics objects.

- Plot objects — Composed of basic graphics objects, but enable properties to be set on plot object level.
- Annotation objects — Exist on a layer separate from other graphics objects.
- Group objects — Create groups of objects that can behave as one in certain respects. You can parent any axes child object (except light) to a group object, including other group object.
- UI objects — User interface objects are used to construct graphical user interfaces.

Graphics objects are interdependent, so the graphics display typically contains a variety of objects that, in conjunction, produce a meaningful graph or picture.

Information on Specific Graphics Objects

See the following sections for more information on the various types of graphics objects:

- "Graphics Windows — the Figure" on page 8-6

- "Core Graphics Objects" on page 8-10
- "Plot Objects" on page 8-19
- "Annotation Objects" on page 8-25
- "Group Objects" on page 8-28
- "Object Properties" on page 8-42

For information on user interface objects and their application, see "Creating Graphical User Interfaces" in the MATLAB documentation.

Graphics Windows — the Figure

In this section...

Introduction

Figures are the windows in which MATLAB software displays graphics. Figures contain menus, toolbars, user-interface objects, context menus, axes and, as axes children, all other types of graphics objects.

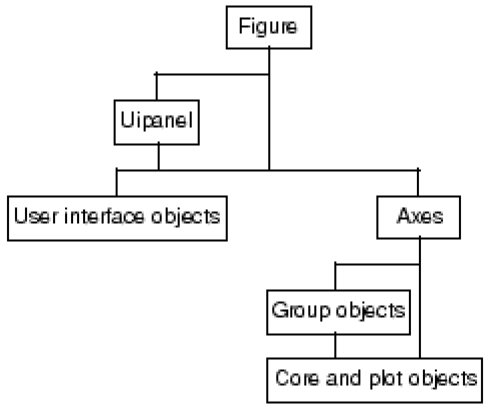
MATLAB places no limits on the number of figures you can create. (Your computer systems might impose limitations, however.)

Figures play two distinct roles in MATLAB:

- Containing data graphs
- Containing graphical user interfaces

Figures can contain both graphs and GUIs components at the same time. For example, a GUI might be designed to plot data and therefore contain an axes as well as user interface objects. See "Example — Using Figure Panels" on page 8-91 for an example of such a GUI.

The following diagram illustrates the types of objects that figures can contain.



Both figures and axes have children that act as containers. A uipanel can contain user interface objects and be parented to a figure and group objects (hggroup and hgtransform) can contain axes children (except light objects) and be parented to an axes.

See "Objects That Can Contain Other Objects" on page 8-89 for more information.

Figures Used for Graphing Data

MATLAB functions that draw graphics (e.g., `plot` and `surf`) create figures automatically if none exist. If there are multiple figures open, one figure is always designated as the "current" figure, and is the target for graphics output.

The `gcf` command returns the handle of the current figure or creates a new figure if one does not exist. For example, enter the following command to see a list of figure properties:

```
get(gcf)
```

The root object `CurrentFigure` > property returns the handle of the current figure, if one exists, or returns empty if there are no figures open:

```
get(0,'CurrentFigure')
ans =
[]
```

See "Controlling Graphics Output" on page 8-70 for more information on how MATLAB determines where to display graphics.

Figure Children for Graphs

Figures that display graphs need to contain axes to provide the frame of reference for objects such as lines and surfaces, which are used to represent data. These data representing objects can be contained in group objects or contained directly in the axes. See "Example — Transforming a Hierarchy of Objects" on page 8-37 for an example of how to use group objects.

Figures can contain multiple axes arranged in various locations within the figure and can be of various sizes. See "Automatic Axes Resize" on page 10-9 and "Multiple Axes per Figure" on page 10-15 for more information on axes.

Figures Used for GUIs

GUIs range from sophisticated applications to simple warning dialog boxes. You can modify many aspects of the figure to fit the intended use by setting figure properties. For example, the following figure properties are often useful when creating GUIs:

- Show or hide the figure menu, while displaying custom-designed menus (`MenuBar`).
- Change the figure title (`Name`).
- Control user access to the figure handle (`HandleVisibility`).
- Create a callback that executes whenever the user resizes the figure (`ResizeFcn`).
- Control display of the figure toolbar (`Toolbar`).
- Assign a context menu (`UIContextMenu`).
- Define callbacks that execute when users click drag or release the mouse over the figure (`WindowButtonDownFcn`, `WindowButtonMotionFcn`, `WindowButtonUpFcn`). See also the `ButtonDownFcn` property.
- Specify whether the figure is modal (`WindowStyle`).

See the Figure Properties reference page for more information on figure characteristics you can specify.

See the "Creating Graphical User Interfaces" documentation for more information about using `figure` to create GUIs.

Root Object — The Figure Parent

The parent of a figure is the root object. You cannot instantiate the root object because its purpose is only to store information. It maintains information on the state of MATLAB, your computer system, and some MATLAB defaults.

There is only one root object, and all other objects are its descendants. You do not create the root object; it exists when you start MATLAB. You can, however, set the values of root properties and thereby affect the graphics display.

For more information, see Root Properties object properties.

More Information on Figures

See the `figure` reference page for information on creating figures.

See "Callback Properties for Graphics Objects" on page 8-108 for information on figure events for which you can define callbacks.

See Chapter 9, "Figure Properties" for information on other figure properties.

Core Graphics Objects

In this section...

Introduction

Core graphics objects include basic drawing primitives:

- Line, text, and polygon shells (patch objects)
- Specialized objects like surfaces, which are composed of a rectangular grid of vertices
- Images
- Light objects, which are not visible but affect the way some objects are colored

Axes contain objects that represent data, such as line, surfaces, contourgroups, etc.

The following table lists the core graphics objects and contains links to the reference pages of the functions used to create each object.

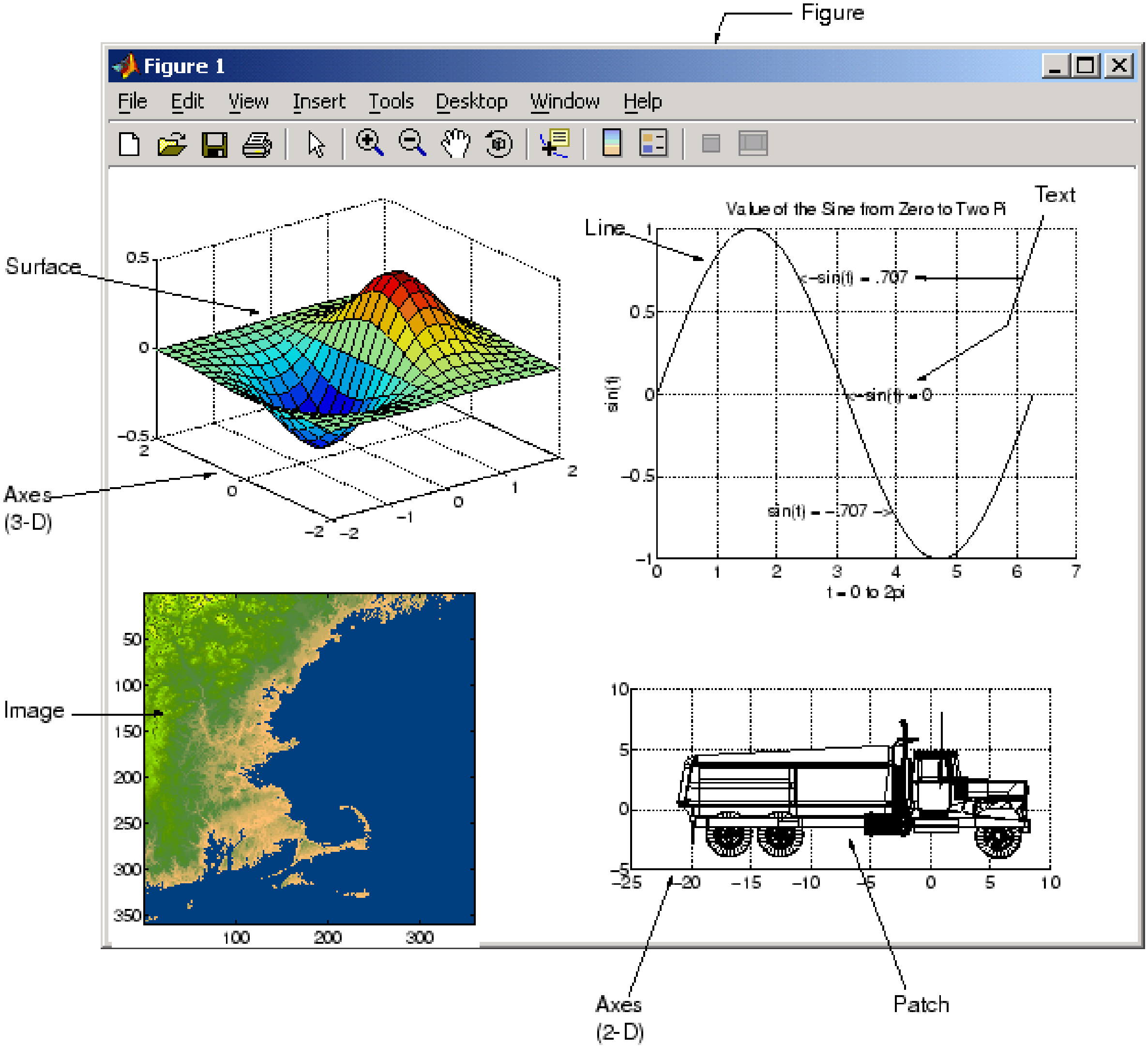
Core Graphics Objects

Function	Purpose
`axes`	Axes objects define the coordinate system for displaying graphs. Axes are always contained within a figure.

Core Graphics Objects (Continued)

Function	Purpose
`image`	2-D representation of a matrix where numeric values are mapped to colors. Images can also be 3-D arrays of RGB values.
`light`	Directional light source located within the axes. Lights affect patches and surfaces, but cannot themselves be seen.
`line`	A line is drawn by connecting the data points that define it.
`patch`	Filled polygons with separate edge properties. A single patch can contain multiple faces, each colored independently with solid or interpolated colors.
`rectangle`	2-D object that has settable edge and face color, and variable curvature (can draw ellipses).
`surface`	3-D grid of quadrilaterals created by plotting the value of each element in a matrix as a height above the *x*-*y* plane.
`text`	Character strings positioned in the coordinate system defined by the axes.

The following picture illustrates some typical core graphics objects.



Core Graphics Objects

This section describes the core graphics objects.

Axes

Axes objects define a frame of reference in a figure window for the display objects that are generally defined by data. For example, MATLAB software creates a line by connecting each data point with a line segment. The axes determines the location of each data point in the figure by defining axis scales (*x*, *y*, and *z*, or radius and angle, etc.)

Axes are children of figures and are parents of core, plot, and group objects.

While annotation objects are also children of axes, they can be parented only to the hidden annotation axes. (See the `annotation` function for more information.)

All functions that draw graphics (e.g., `plot`, `surf`, `mesh`, and `bar`) create an axes object if one does not exist. If there are multiple axes within the figure, one axes is always designated as the “current” axes, and is the target for display of the above-mentioned graphics objects. (Uicontrols and uimenus are not children of axes.)

Image

A MATLAB image consists of a data matrix and possibly a colormap. There are three basic image types that differ in the way that data matrix elements are interpreted as pixel colors—indexed, intensity, and truecolor. Since images are strictly 2-D, you can view them only at the default 2-D view.

Light

Light objects define light sources that affect all patch and surface objects within the axes. You cannot see lights, but you can set properties that control the style of light source, color, location, and other properties common to all graphics objects.

Line

Line objects are the basic graphics primitives used to create most 2-D and some 3-D plots. High-level functions `plot`, `plot3`, and `loglog` (and others)

create line objects. The coordinate system of the axes positions and orients the line.

Patch

Patch objects are filled polygons with edges. A single patch can contain multiple faces, each colored independently with solid or interpolated colors. `fill`, `fill3`, and `contour3` create patch objects. The coordinate system of the axes positions and orients the patch.

Rectangle

Rectangle objects are 2-D filled areas having a shape that can range from a rectangle to an ellipse. Rectangles are useful for creating flowchart-type drawings.

Surface

Surface objects are 3-D representations of matrix data created by plotting the value of each matrix element as a height above the *x-y* plane. Surface plots are composed of quadrilaterals whose vertices are specified by the matrix data. MATLAB can draw surfaces with solid or interpolated colors or with only a mesh of lines connecting the points. The coordinate system of the axes positions and orients the surface.

The high-level function `pcolor` and the `surf` and `mesh` group of functions create surface objects.

Text

Text objects are character strings. The coordinate system of the parent axes positions the text. The high-level functions `title`, `xlabel`, `ylabel`, `zlabel`, and `gtext` create text objects.

Example — Creating Core Graphics Objects

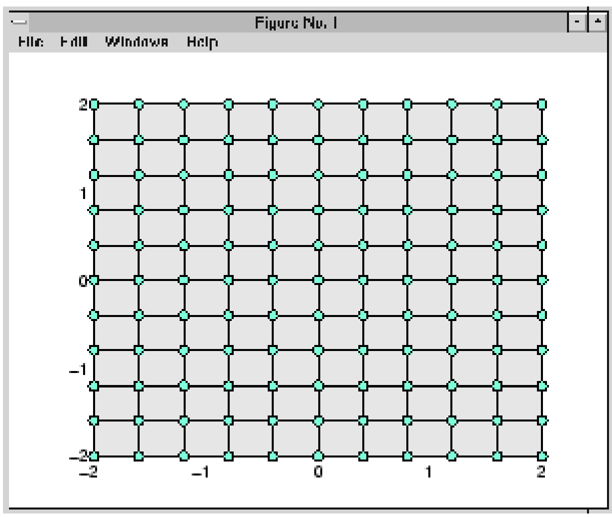
Object creation functions have a syntax of the form

```
handle = function('propertyname',propertyvalue,...)
```

You can specify a value for any object property (except those that are read only) by passing property name/value pairs as arguments. The function returns the handle of the object it creates, which you can use to query and modify properties after creating the object.

This example evaluates a mathematical function and creates three graphics objects using the property values specified as arguments to the `figure`, `axes`, and `surface` commands. MATLAB uses default values for all other properties.

```
[x,y] = meshgrid([-2:.4:2]);
Z = x.*exp(-x.^2-y.^2);
fh = figure('Position',[350 275 400 300],'Color','w');
ah = axes('Color',[.8 .8 .8],'XTick',[-2 -1 0 1 2],...
          'YTick',[-2 -1 0 1 2]);
sh = surface('XData',x,'YData',y,'ZData',Z,...
             'FaceColor',get(ah,'Color')+.1,...
             'EdgeColor','k','Marker','o',...
             'MarkerFaceColor',[.5 1 .85]);
```

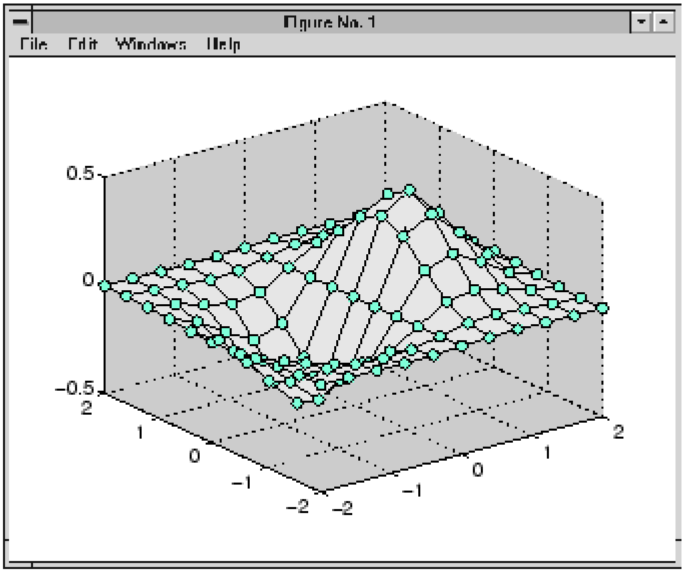


The `surface` function does not use a 3-D view like the high-level `surf` functions. Object creation functions simply add new objects to the current axes without changing axes properties, except the `Children` property, which

now includes the new object and the axis limits (XLim, YLim, and ZLim), if necessary.

You can change the view using the camera commands or use the view command.

```
view(3)
```



Parenting

By default, all statements that create graphics objects do so in the current figure and the current axes (if the object is an axes child). However, you can specify the parent of an object when you create it. For example,

```
axes('Parent',figure_handle,...)
```

creates an axes in the figure identified by figure_handle. You can also move an object from one parent to another by redefining its Parent property:

```
set(gca,'Parent',figure_handle)
```

High-Level Versus Low-Level Functions

Many MATLAB graphics functions call the object creation functions to draw graphics objects. However, high-level routines also clear the axes or create a new figure, depending on the settings of the axes and figure `NextPlot` properties.

In contrast, core object creation functions simply create their respective graphics objects and place them in the current parent object. They do not respect the settings of the figure or axes `NextPlot` property.

For example, if you call the `line` function,

```
line('XData',x,'YData',y,'ZData',z,'Color','r')
```

MATLAB draws a red line in the current axes using the specified data values. If there is no axes, MATLAB creates one. If there is no figure window in which to create the axes, MATLAB creates it as well.

If you call the `line` function a second time, MATLAB draws the second line in the current axes without erasing the first line. This behavior is different from high-level functions like `plot` that delete graphics objects and reset all axes properties (except `Position` and `Units`). You can change the behavior of high-level functions by using the `hold` command or by changing the setting of the axes `NextPlot` property.

See "Controlling Graphics Output" on page 8-70 for more information on this behavior and on using the `NextPlot` property.

Simplified Calling Syntax

Object creation functions have convenience forms that allow you to use a simpler syntax. For example,

```
text(.5,.5,.5,'Hello')
```

is equivalent to

```
text('Position',[.5 .5 .5],'String','Hello')
```

Using the convenience form of an object creation function can cause subtle differences in behavior when compared to formal property name/property value syntax.

A Note About Property Names

By convention, MATLAB documentation capitalizes the first letter of each word that makes up a property name, such as `LineStyle` or `XTickLabelMode`. While this makes property names easier to read, MATLAB does not check for uppercase letters. In addition, you need to use only enough letters to identify the name uniquely, so you can abbreviate most property names.

In M-files, however, using the full property name can prevent problems with futures releases of MATLAB if a shortened name is no longer unique because of the addition of new properties.

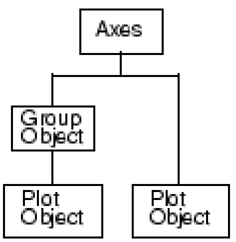
Plot Objects

In this section...

Introduction

A number of high-level plotting functions create plot objects. The properties of plot objects provide easy access to the important properties of the core graphics objects that the plot objects contain.

Plot object parents can be axes or group objects (`hggroup` or `hgtransform`). See "Objects That Can Contain Other Objects" on page 8-89 for examples.



This table lists the plot objects and the graphing functions that use them. Click the object names to see a description of their properties.

Plot Objects

Object	Purpose
`areaseries`	Used to create `area` graphs.
`barseries`	Used to create `bar` graphs.
`contourgroup`	Used to create `contour` graphs.
`errorbarseries`	Used to create `errorbar` graphs.

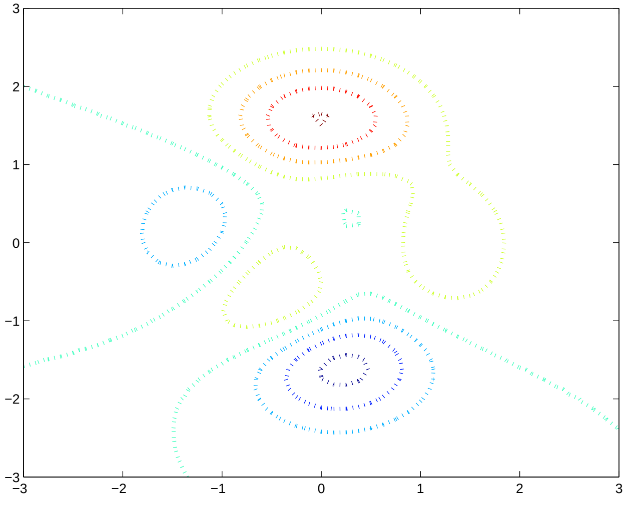
Plot Objects (Continued)

Object	Purpose
lineseries	Used by line plotting functions (plot, plot3, etc.).
quivergroup	Used to create quiver and quiver3 graphs.
scattergroup	Used to create scatter and scatter3 graphs.
stairseries	Used to create stairstep graphs (stairs).
stemseries	Used to create stem and stem3 graphs.
surfaceplot	Used by the surf and mesh group of functions.

Creating a Plot Object

For example, the following statements create a contour graph of the peaks function and then set the line style and width of the contour lines:

```
[x,y,z] = peaks;
[c,h] = contour(x,y,z);
set(h,'LineWidth',3,'LineStyle',':')
```



The contour plot object enables you to set the line width and style of the contour graph by setting two properties. Looking at the core objects contained in the contour plot object reveals a number of patch objects whose edges are used to implement the contour line, which you would otherwise need to set individually.

```
child_handles = get(h,'Children');
get(child_handles,'Type')
ans =
    'patch'
    'patch'
    'patch'
    'patch'
    'patch'
    'patch'
    'patch'
    'patch'
    'patch'
    'patch'
    'patch'
    'patch'
```

Identifying Plot Objects Programmatically

Plot objects all return `hggroup` as the value of the `Type` property. If you want to be able to identify plot objects programmatically but do not have access to the object's handle, set a value for the object's `Tag` property.

For example, the following statements create a bar graph with five `barseries` objects and assign a different value for the `Tag` property on each object:

```
h = bar(rand(5));
set(h,{'Tag'},{'bar1','bar2','bar3','bar4','bar5'}')
```

The cell array of property values must be transposed (') to have the proper shape. See the `set` function for more information on setting properties.

No User Default Values

You cannot define default values for plot objects.

Plot Objects and Backward Compatibility

Note The `v6` option discussed in this section is now obsolete and will be removed in a future version of MATLAB software.

Plotting functions that create plot objects can introduce incompatibilities with code written before MATLAB Version 7.x. However, all plotting functions that return handles to plot objects support an optional argument (`'v6'`) that forces the functions to use core objects, as was the case in MATLAB software before Version 7.

- See “Plot Objects” on page 8-19 for a list of functions that create plot objects.
- See “Core Graphics Objects” on page 8-10 for a list of core graphics objects.

Saving Figures That Are Compatible with Previous Version of MATLAB

Create backward-compatible FIG-files by following these two steps:

- Ensure that any plotting functions used to create the contents of the figure are called with the `'v6'` argument, where applicable.
- Use the `'-v6'` option with the `hgsave` command.

For example:

```
h = figure;
t = 0:pi/20:2*pi;
plot('v6',t,sin(t).*2)
hgsave(h,'myFigFile','-v6')
```

You can set a general MATLAB preference to ensure that figures saved by selecting **File>Save** are backward compatible. To access MATLAB preferences, select **Preferences** from the Desktop **File** menu. Expand the **General** node and select **MAT Files**. Click **Ensure backward compatibility (-v6)**. This setting affects all FIG-files and MAT-files that you create.

Linking Graphs to Variables — Data Source Properties

In this section...

Introduction

Plot objects enable you to link a MATLAB expression with properties that contain data. For example, the `lineseries` object has data source properties associated with the `XData`, `YData`, and `ZData` properties. These properties are called `XDataSource`, `YDataSource`, and `ZDataSource`.

To use a data source property:

1. Assign the name of a variable to the data source property that you want linked to an expression.
2. Calculate a new value for the variable.
3. Call the `refreshdata` function to update the plot object data.

`refreshdata` enables you to specify whether to use a variable in the base workspace or the workspace of the function from which you call `refreshdata`.

Data Source Example

The following example illustrates how to use this technique:

```
function datasource_ex
t = 0:pi/20:2*pi;
y = exp(sin(t));
h = plot(t,y,'YDataSource','y');
for k = 1:.1:10
 y = exp(sin(t.*k));
 refreshdata(h,'caller') % Evaluate y in the function workspace
 drawnow; pause(.1)
end
```

Changing the Size of Data Variables

If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Annotation Objects

In this section...

Introduction

Users typically create annotation objects from the Plot Edit toolbar or the **Insert** menu (select **Plot Edit** in the **View** menu to display the Plot Edit toolbar). However, you can also create annotation objects using the `annotation` function.

Annotation objects are created in a hidden axes that extends the full width and height of the figure. This enables you to specify the locations of annotation objects anywhere in the figure using normalized coordinates (the lower-left corner is the point 0,0, the upper-right corner is the point 1,1).

Annotation Object Properties

Note Doot change any of the properties of the annotation axes or parent any graphics objects to this axes. Use the `annotation` function or the graphics tools to create annotation objects.

The following links access descriptions of the properties you can set on the respective annotation objects:

- Annotation arrow properties
- Annotation doublearrow properties
- Annotation ellipse properties
- Annotation line properties
- Annotation rectangle properties

- Annotation textarrow properties
- Annotation textbox properties

To modify the appearance of annotation objects created with the plotting tools, use "The Property Editor" on page 1-28.

Example — Enclosing Subplots with an Annotation Rectangle

The following example shows how to create a rectangle annotation object and use it to highlight two subplots in a figure. This example uses the axes properties `Position` and `TightInset` to determine the location and size of the annotation rectangle.

1 Create an array of subplots:

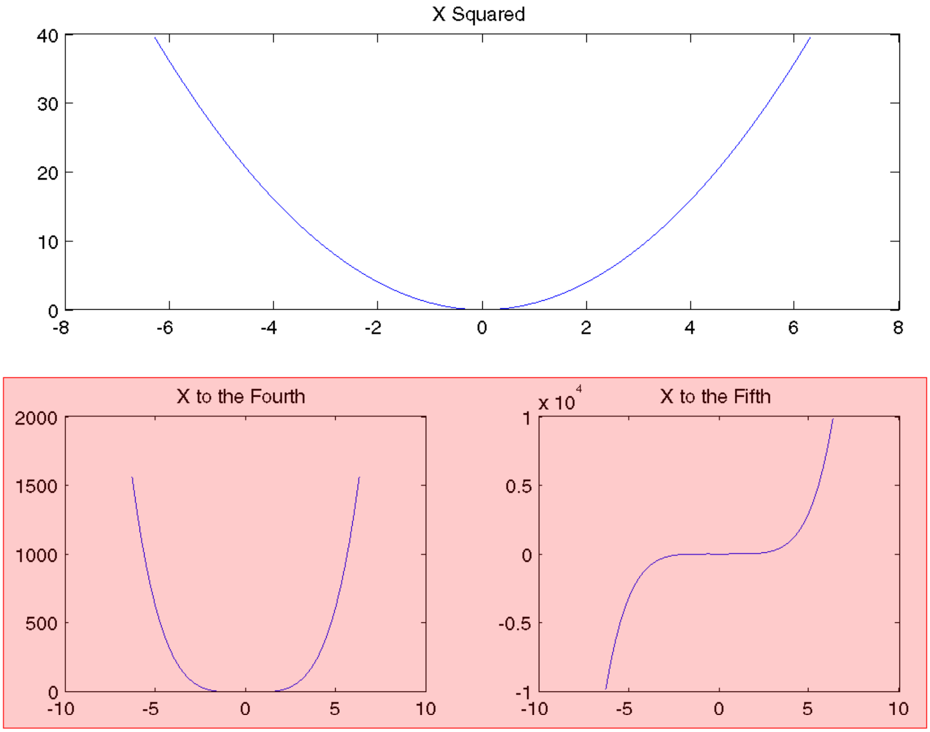
```
x = -2*pi:pi/12:2*pi;
y = x.^2;
subplot(2,2,1:2)
plot(x,y)
h1=subplot(223);
y = x.^4;
plot(x,y)
h2=subplot(224);
y = x.^5;
plot(x,y)
```

2 Determine the location and size of the annotation rectangle required to enclose axes, tick mark labels, and title using the axes `Position` and `TightInset` properties:

```
p1 = get(h1,'Position');
t1 = get(h1,'TightInset');
p2 = get(h2,'Position');
t2 = get(h2,'TightInset');
x1 = p1(1)-t1(1); y1 = p1(2)-t1(2);
x2 = p2(1)-t2(1); y2 = p2(2)-t2(2);
w = x2-x1+t1(1)+p2(3)+t2(3); h = p2(4)+t2(2)+t2(4);
```

3 Create the annotation rectangle to enclose the lower two subplots. Make the rectangle a translucent red with a solid border:

```
annotation('rectangle',[x1,y1,w,h],...
 'FaceAlpha',.2,'FaceColor','red','EdgeColor','red');
```



Group Objects

In this section...

Introduction

Group objects enable you to treat a number of axes child objects as one group. For example, you can make the entire group visible or invisible, select all objects when only one is clicked, or apply a transform matrix to reposition the objects by setting only one property on the group object.

There are two group objects:

- hggroup — Use when you want to create a group of objects and control the visibility or selectability of the group based on what happens to any individual object in the group. Create hggroup objects with the `hggroup` function.
- hgtransform — Use when you want to transform a group of objects. Transforms include rotation, translation, scaling, etc. See "Example — Transforming a Hierarchy of Objects" on page 8-37 for an example. Create hgtransform objects with the `hgtransform` function.

The difference between the hggroup and hgtransform objects is the ability of the hgtransform object to apply a transform matrix (via its `Matrix` property) to all objects for which it is the parent.

Note You cannot parent light objects to hggroup or hgtransform objects.

Creating a Group

You create a group by parenting axes children to an hggroup or hgtransform object:

```
hb = bar(rand(5)); % creates 5 barseries objects
hg = hggroup;
set(hb,'Parent',hg) % parent the barseries to the hggroup
set(hg,'Visible','off') % makes all barseries invisible
```

Group objects can be the parent of any number of axes children, including other group objects.

Note Many plotting functions clear the axes (i.e., remove axes children) before drawing the graph. Clearing the axes also deletes any hggroup or hgtransform objects in the axes.

Transforming Objects

The hgtransform object's `Matrix` property enables you to apply a transform to all the hgtransform's children in unison. Typical transforms include rotation, translation, and scaling. You define a transform with a four-by-four transformation matrix, which is described in the following sections.

Creating a Transform Matrix

The `makehgtform` function simplifies the construction of matrices to perform rotation, translation, and scaling. See the "Example — Transforming a Hierarchy of Objects" on page 8-37 section for information on creating transform matrices using `makehgtform`.

Rotation

Rotation transforms rotate objects about the -, *y*-, or *z*-axis, with positive angles rotating counterclockwise while sighting along the respective axis toward the origin. If the desired angle of rotation is [[THETA]], the following matrices define this rotation about the respective axis.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x & 0 \\ 0 & \sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To create a transform matrix for rotation about an arbitrary axis, use the `makehgtform` function.

Translation

Translation transforms move objects with respect to their current locations. Specify the translation as distances t_x, t_y, and t_z in data space units. The following matrix shows the location of these elements in the transform matrix.

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

Scaling transforms change the sizes of objects. Specify scale factors s_x,s_y, and s_z and construct the following matrix:

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

You cannot use scale factors less than or equal to zero.

The Default Transform

The default transform is the identity matrix, which you can create with the `eye` function. Here is the identity matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

See “Undoing Transform Operations” on page 8-33 for related information.

Disallowed Transforms: Perspective

Perspective transforms change the distance at which you view an object. The following matrix is an example of a perspective transform matrix, which Handle Graphics does not allow:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & p_x & 0 \end{bmatrix}$$

In this case, p_y is the perspective factor.

Disallowed Transforms: Shear

Shear transforms keep all points along a given line (or plane, in 3-D coordinates) fixed while shifting all other points parallel to the line (plane) proportional to their perpendicular distance from the fixed line (plane). The following matrix is an example of a shear transform matrix, which Handle Graphics does not allow:

$$\begin{bmatrix} 1 & s_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In this case, s_y is the shear factor and can replace any zero element in an identity matrix.

Absolute vs. Relative Transforms

Transforms are specified in absolute terms, not relative to the current transform. For example, if you apply a transform that translates the hgtransform object 5 units in the *x* direction and then you apply another transform that translates it 4 units in the *y* direction, the resulting position of the object is 4 units in the *y* direction from its original position.

If you want transforms to accumulate, you must concatenate the individual transforms into a single matrix. See "Combining Transforms into One Matrix" on page 8-32 for more information.

Combining Transforms into One Matrix

It is usually more efficient to combine various transform operations into one matrix by concatenating (multiplying) the individual matrices and setting the `Matrix` property to the result. Matrix multiplication is not commutative, so the order in which you multiply the matrices affects the result. For example, suppose you want to perform an operation that scales, translates, and then rotates. You multiply the matrices as follows:

```
C = R*T*S % operations are performed from right to left
```

where `S` is the scaling matrix, `T` is the translation matrix, `R` is the rotation matrix, and `C` is the composite of the three operations. You then set the hgtransform object's `Matrix` property to `C`:

```
set(hgtransform_handle,'Matrix',C)
```

The following sets of statements are not equivalent:

```
set(hgtransform_handle,'Matrix',C)% Transform as above
set(hgtransform_handle,'Matrix',eye(4) ) % Undo transform
```

versus

```
C = eye(4)*R*T*S % Multiply identity matrix as last step
set(hgtransform_handle,'Matrix',C)
```

Concatenating the identity matrix to other matrices has no effect on the composite matrix.

Undoing Transform Operations

Since transform operations are specified in absolute terms (not relative to the current transform), you can undo a series of transforms by setting the current transform to the identity matrix. For example,

```
set(hgtransform_handle,'Matrix',eye(4))
```

returns the object *hgtransform_handle* to its untransformed orientation.

Rotations Away From the Origin

Since rotations are performed about the origin, it is often necessary to translate the hgtransform object so that the desired axis of rotation is temporarily at the origin. After applying the rotation transform matrix, you then translate the hgtransform object back to its original position. The following example illustrates how to do this.

Suppose you want to rotate a surface about the *y*-axis at the center of the surface (the *y*-axis that passes through the point `x = 20` in this example).

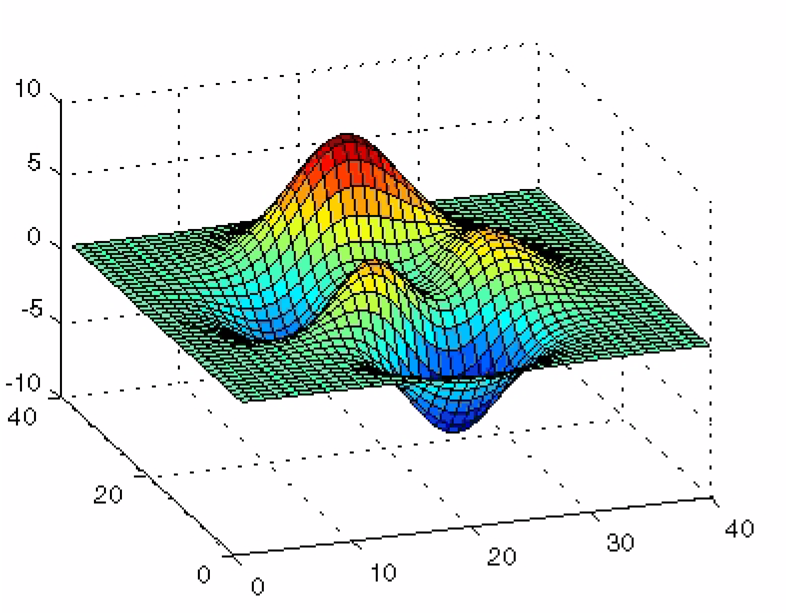
Note If you are using the MATLAB Help browser, you can run this example or open it in the MATLAB editor.

---

1 Create a surface and an hgtransform object. Parent the surface to the hgtransform object:

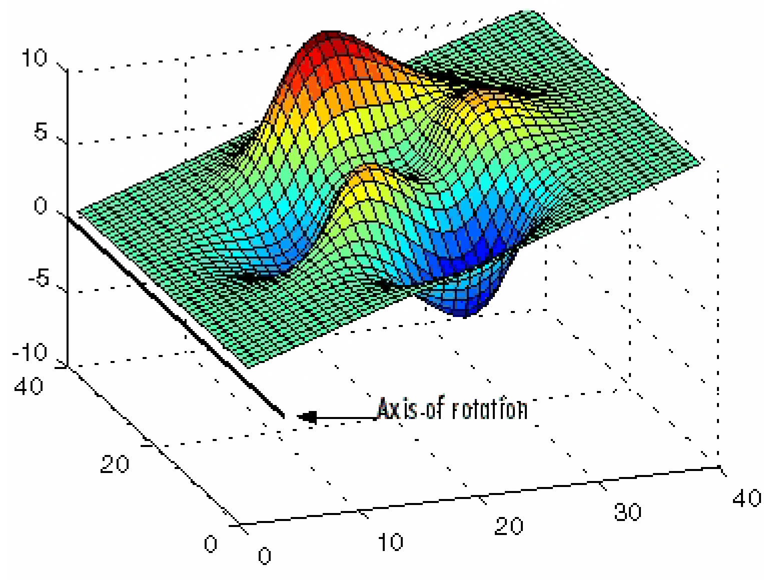
```
h = surf(peaks(40)); view(-20,30)
t = hgtransform;
set(h,'Parent',t)
```

The following picture shows the surface.



2 Create and set a *y*-axis rotation matrix to rotate the surface by –15 degrees:

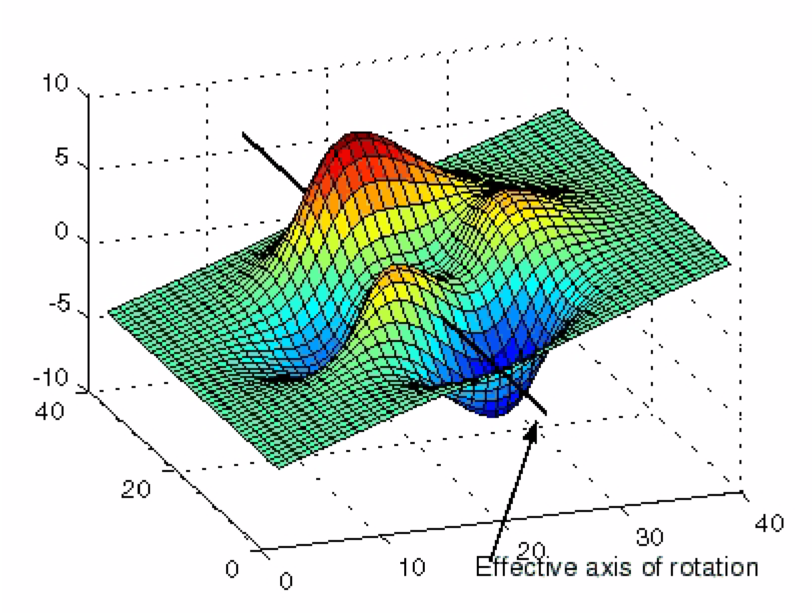
```
ry_angle = -15*pi/180; % Convert to radians
Ry = makehgtform('yrotate',ry_angle);
set(t,'Matrix',Ry)
```



Notice that the surface rotated –15 degrees about the *y*-axis that passes through the origin. However, to rotate about the *y*-axis that passes through the point `x = 20`, you must translate the surface in *x* by 20 units.

3 Create two translation matrices, one to translate the surface –20 units in *x* and another to translate 20 units back. Concatenate the two translation matrices with the rotation matrix in the correct order and set the transform:

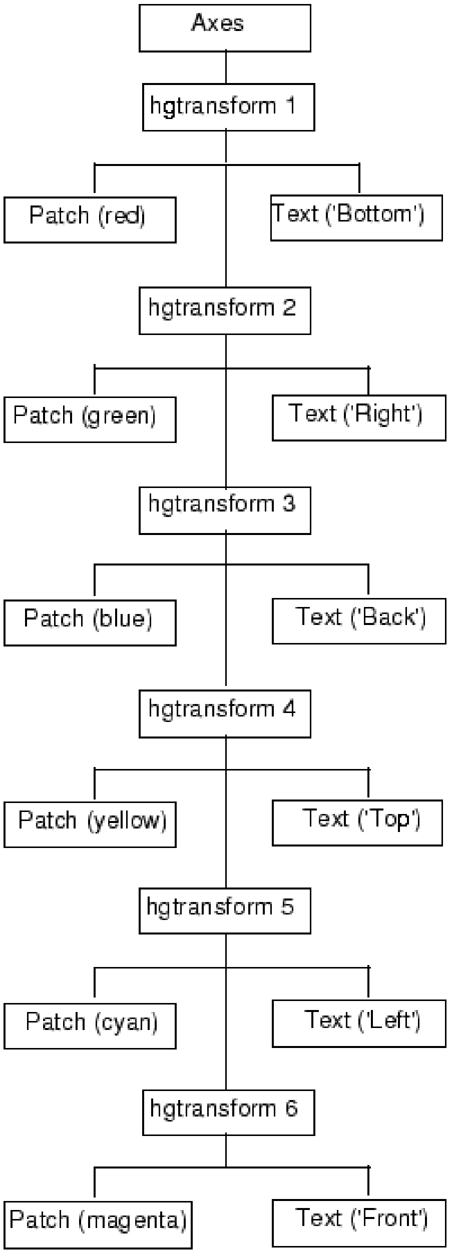
```
Tx1 = makehgtform('translate',[-20 0 0]);
Tx2 = makehgtform('translate',[20 0 0]);
set(t,'Matrix',Tx2*Ry*Tx1)
```


10
5
0
-5
-10
40
20
0
0
10
20
30
40
Effective axis of rotation

Example — Transforming a Hierarchy of Objects

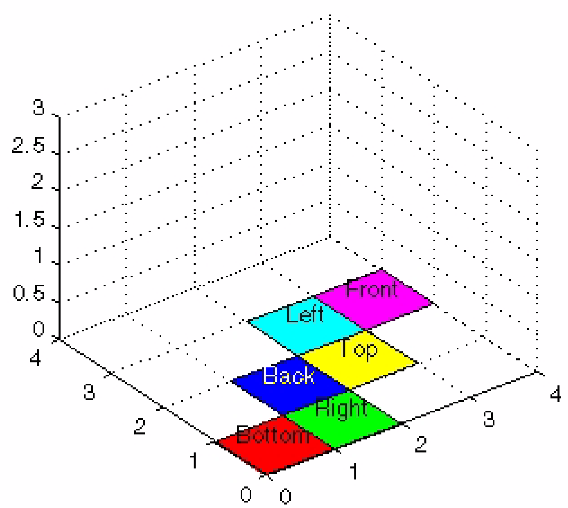
This example creates a hierarchy of hgtransform objects, which are then transformed in sequence to create a cube from six squares. The example illustrates how you can parent hgtransform objects to other hgtransform objects to create a hierarchy and how transforming members of a hierarchy affects subordinate members.

The following picture illustrates the hierarchy.



The diagram on the left represents the object hierarchy in the picture below.

Through a series of simple rotations and translations, the six squares are folded into a cube.



Note If you are using the MATLAB Help browser, you can run this example or open it in the MATLAB Editor.

---

1 Set the figure `Renderer` property to `zbuffer` so MATLAB uses double buffering to prevent flashing during the loop. Set up the figure and the view:

```
set(gcf,'Renderer','zbuffer');
% Set axis limits and view
set(gca,'XLim',[0 4], 'YLim',[0 4], 'ZLim', [0 3]);
view(3); axis equal; grid on
```

2 Define a hierarchy of hgtransform objects:

```
t(1) = hgtransform;
t(2) = hgtransform('parent',t(1));
t(3) = hgtransform('parent',t(2));
t(4) = hgtransform('parent',t(3));
t(5) = hgtransform('parent',t(4));
t(6) = hgtransform('parent',t(5));
```

3 Create the patch and text objects and parent each pair to the respective hgtransform object.

The data defining each patch object and the locations of all text objects are the same and are assigned by a single call to `set`. The objects are then translated to the desired positions on screen.

```
% Patch data
X = [0 0 1 1];
Y = [0 1 1 0];
Z = [0 0 0 0];
% Text data
Xtext = .5;
Ytext = .5;
Ztext = .15;
% Parent corresponding pairs of objects (patch and text)
% into the object hierarchy
p(1) = patch('FaceColor','red','Parent',t(1));
```

```
txt(1) = text('String','Bottom','Parent',t(1));
p(2) = patch('FaceColor','green','Parent',t(2));
txt(2) = text('String','Right','Parent',t(2));
p(3) = patch('FaceColor','blue','Parent',t(3));
txt(3) = text('String','Back','Color','white','Parent',t(3));
p(4) = patch('FaceColor','yellow','Parent',t(4));
txt(4) = text('String','Top','Parent',t(4));
p(5) = patch('FaceColor','cyan','Parent',t(5));
txt(5) = text('String','Left','Parent',t(5));
p(6) = patch('FaceColor','magenta','Parent',t(6));
txt(6) = text('String','Front','Parent',t(6));
% Set the patch x, y, and z data
set(p,'XData',X,'YData',Y,'ZData',Z)
% Set the position and alignment of the text
set(txt,'Position',[Xtext Ytext Ztext],...
    'HorizontalAlignment','center',...
    'VerticalAlignment','middle')
```

4 Translate the squares (patch objects) to the desired locations. As hgtransform object 2 is translated, all its children (including hgtransform objects 3 through 6) are also translated. Therefore, each translation requires moving the square by only one unit in either the *x* or *y* direction. hgtransform object 1 is left at its original position.

```
% Set up initial translation transform matrices
% Translate 1 unit in x
Tx = makehgtform('translate',[1 0 0]);
% Translate 1 unit in y
Ty = makehgtform('translate',[0 1 0]);
% Set the Matrix property of each hgtransform object (2-6)
set(t(2),'Matrix',Tx);
drawnow
set(t(3),'Matrix',Ty);
drawnow
set(t(4),'Matrix',Tx);
drawnow
set(t(5),'Matrix',Ty);
drawnow
set(t(6),'Matrix',Tx);
```

5 Specify the rotation angle and perform transforms:

```
% Specify rotation angle (pi/2 radians = 90 degrees)
fold = pi/2;
% Rotate -y, translate x
Ry = makehgtform('yrotate',-fold);
RyTx = Tx*Ry;
% Rotate x, translate y
Rx = makehgtform('xrotate',fold);
RxTy = Ty*Rx;
```

6 Set the transforms and draw after each group transform with a slight pause

```
set(t(6),'Matrix',RyTx);
pause(.5)
set(t(5),'Matrix',RxTy);
pause(.5)
set(t(4),'Matrix',RyTx);
pause(.5)
set(t(3),'Matrix',RxTy);
pause(.5)
set(t(2),'Matrix',RyTx);
pause(.5)
```

Object Properties

In this section...

Introduction

A graphics object's properties control many aspects of its appearance and behavior. Properties include general information such as the object's type, its parent and children, and whether it is visible, as well as information unique to the particular class of object.

For example, from any given figure object you can obtain the identity of the last key pressed in the window, the location of the pointer, or the handle of the most recently selected menu.

Note The simplest way to access the documentation of all object properties is using the Handle Graphics Property Browser.

Storing Object Information

MATLAB software organizes graphics information into a hierarchy and stores information about objects in properties. For example, root properties contain the handle of the current figure and the current location of the pointer (cursor), figure properties maintain lists of their descendants and keep track of certain events that occur within the window, and axes properties contain information about how each child object uses the figure colormap and the color order used by the `plot` function.

Changing Values

You can query the current value of any property and specify most property values (although some are set by MATLAB and are read only). Property values apply uniquely to a particular instance of an object; setting a value for one object does not change this value for other objects of the same type.

Order Dependence of Setting Property Values

MATLAB sets the values of properties in the order in which properties are assigned values in a statement. For example, the following calls to the `figure` function create very different results. This statement,

```
figure('Position',[1 1 400 300],'Units','inches')
```

creates a figure in the lower-left corner of the screen that is 400 pixels in width and 300 pixels in height. If you reverse the order of the `Position` and `Units` properties, MATLAB creates a figure that is too large to display (400 by 300 inches):

```
figure('Units','inches','Position',[1 1 400 300])
```

Properties Are Interpreted from Left to Right

In the first figure above, MATLAB creates a figure of the specified size using the default `Units` (pixels) and then sets the `Units` to inches. In the second case MATLAB sets the `Units` to inches and uses these units to interpret the specified figure `Position`. MATLAB interprets the property values from left to right:

```
set(gcf,'Units','pixels')
get(gcf,'Position')
ans =
  1.0e+004 *
    0.0097    2.7760    0.1924    0.1137
% Change the Units, set the Position,
% and change Units again in one statement
set(gcf,'Units','pixels','Position',[1 1 400 300],'Units',...
  'inches')
get(gcf,'Position')
ans =
         0         0    4.1667    3.1250
```

Default Values

You can set default values that affect all subsequently created objects. Whenever you do not define a value for a property, either as a default or when you create the object, MATLAB uses “factory-defined” values.

Plot objects do not allow you to set default values.

The reference entry for each object creation function provides a complete list of the properties associated with the graphics object.

Properties Common to All Objects

Some properties are common to all graphics objects, as illustrated in the following table.

Property	Description
`BeingDeleted`	Has a value of `on` when object’s `DeleteFcn` has been called.
`BusyAction`	Controls the way MATLAB handles callback routine interruption defined for the particular object.
`ButtonDownFcn`	Callback routine that executes when button press occurs.
`Children`	Handles of all this object’s child objects.
`Clipping`	Mode that enables or disables clipping (meaningful only for axes children).
`CreateFcn`	Callback routine that executes when this type of object is created.
`DeleteFcn`	Callback routine that executes when you issue a command that destroys the object.
`HandleVisibility`	Allows you to control the availability of the object’s handle from the command line and from within callback routines.
`HitTest`	Determines if object can become the current object when selected by a mouse click.
`Interruptible`	Determines whether a callback routine can be interrupted by a subsequently invoked callback routine.
`Parent`	The object’s parent.
`Selected`	Indicates whether object is selected.

Property	Description
`SelectionHighlight`	Specifies whether object visually indicates the selection state.
`Tag`	User-specified object label.
`Type`	The type of object (figure, line, text, etc.).
`UserData`	Any data you want to associate with the object.
`Visible`	Determines whether or not the object is visible.

Setting and Querying Property Values

In this section...

Using set and get

The `set` and `get` functions specify and retrieve the value of existing graphics object properties. They also enable you to list possible values for properties that have a fixed set of values. (You can also use the Property Editor to set many property values.)

The basic syntax for setting the value of a property on an existing object is

```
set(object_handle,'PropertyName','NewPropertyValue')
```

To query the current value of a specific object's property, use a statement like

```
returned_value = get(object_handle,'PropertyName');
```

Property names are always quoted strings. Property values depend on the particular property.

See "Accessing Object Handles" on page 8-59 and the `findobj` command for information on finding the handles of existing objects.

Setting Property Values

Change the properties of an existing object using the `set` function and the handle returned by the creating function. For example, this statement moves the *y*-axis to the right side of the plot on the current axes:

```
set(gca,'YAxisLocation','right')
```

If the handle argument is a vector, MATLAB software sets the specified value on all identified objects.

Specify property names and property values using structure arrays or cell arrays. This can be useful if you want to set the same properties on a number of objects. For example, define a structure to set axes properties appropriately to display a particular graph:

```
view1.CameraViewAngleMode = 'manual';
view1.DataAspectRatio = [1 1 1];
view1.ProjectionType = 'Perspective';
```

To set these values on the current axes, type

```
set(gca,view1)
```

Listing Possible Values

Use `set` to display the possible values for many properties without actually assigning a new value. For example, this statement obtains the values you can specify for line object markers:

```
set(obj_handle,'Marker')
```

MATLAB returns a list of values for the `Marker` property for the type of object specified by `obj_handle`. Braces indicate the default value:

```
[ + | o | * | . | x | square | diamond | v | ^ | > | < |
pentagram | hexagram | {none} ]
```

To see a list of all settable properties along with possible values of properties that accept string values, use `set` with just an object handle:

```
set(object_handle)
```

For example, for a surface object, MATLAB returns

```
CData
CDataScaling: [ {on} | off]
EdgeColor: [ none | {flat} | interp ] ColorSpec.
EraseMode: [ {normal} | background | xor | none ]
FaceColor: [ none | {flat} | interp | texturemap ] ColorSpec.
LineStyle: [ {-} | -- | : | -. | none ]
   .
   .
```

```
  .
Visible: [ {on} | off ]
```

If you assign the output of the set function to a variable, MATLAB returns the output as a structure array:

```
a = set(gca);
```

The field names in a are the object's property names and the field values are the possible values for the associated property:

```
a.GridLineStyle
ans =

    '-'
    '--'
    ':'
    '-.'
    'none'
```

returns the possible values for the axes grid line styles. While property names are not case sensitive, MATLAB structure field names are:

```
a.gridlinestyle
??? Reference to non-existent field 'gridlinestyle'.
```

returns an error.

Querying Property Values

Use get to query the current value of a property or of all the object's properties. For example, check the value of the current axes PlotBoxAspectRatio property:

```
get(gca,'PlotBoxAspectRatio')
ans =
     1     1     1
```

MATLAB lists the values of all properties, where practical. However, for properties containing data, MATLAB lists the dimensions only (for example, CurrentPoint and ColorOrder).

```
AmbientLightColor = [1 1 1]
Box = off
CameraPosition = [0.5 0.5 2.23205]
CameraPositionMode = auto
CameraTarget = [0.5 0.5 0.5]
CameraTargetMode = auto
CameraUpVector = [0 1 0]
CameraUpVectorMode = auto
CameraViewAngle = [32.2042]
CameraViewAngleMode = auto
CLim: [0 1]
CLimMode: auto
Color: [0 0 0]
CurrentPoint: [ 2x3  double]
ColorOrder: [ 7x3  double]
  .
  .
  .
Visible = on
```

Querying Individual Properties

You can obtain the data from the property by getting that property individually:

```
get(gca,'ColorOrder')
ans =
         0         0    1.0000
         0    0.5000         0
    1.0000         0         0
         0    0.7500    0.7500
    0.7500         0    0.7500
    0.7500    0.7500         0
    0.2500    0.2500    0.2500
```

Returning a Structure

If you assign the output of `get` to a variable, MATLAB creates a structure array whose field names are the object property names and whose field values are the current values of the named property.

For example, if you plot some data, x and y,

```
h = plot(x,y);
```

and get the properties of the line object created by plot,

```
a = get(h);
```

you can access the values of the line properties using the field name. This call to the text command places the string 'x and y data' at the first data point and colors the text to match the line color:

```
text(x(1),y(1),'x and y data','Color',a.Color)
```

If x and y are matrices, plot draws one line per column. To label the plot of the second column of data, reference that line:

```
text(x(1,2),y(1,2),'Second set of data','Color',a(2).Color)
```

Querying Groups of Properties

You can define a cell array of property names and conveniently use it to obtain the values for those properties. For example, suppose you want to query the values of the axes “camera mode” properties. First, define the cell array:

```
camera_props(1) = {'CameraPositionMode'};
camera_props(2) = {'CameraTargetMode'};
camera_props(3) = {'CameraUpVectorMode'};
camera_props(4) = {'CameraViewAngleMode'};
```

Use this cell array as an argument to obtain the current values of these properties:

```
get(gca,camera_props)
ans =
     'auto' 'auto' 'auto' 'auto'
```

Factory-Defined Property Values

MATLAB software defines values for all properties, which are used if you do not specify values as arguments or as defaults. You can obtain a list of all factory-defined values with the statement

```
a = get(0,'Factory');
```

`get` returns a structure array whose field names are the object type and property name concatenated, and field values are the factory value for the indicated object and property. For example, this field,

```
UimenuSelectionHighlight: 'on'
```

indicates that the factory value for the `SelectionHighlight` property on uimenu objects is `on`.

You can get the factory value of an individual property with

```
get(0,'FactoryObjectTypePropertyName')
```

For example:

```
get(0,'FactoryTextFontName')
```

Setting Default Property Values

In this section...

Factory- and User-Defined Values

All object properties have values built into MATLAB software (i.e., factory-defined values). You can also define your own default values at any point in the object hierarchy.

You cannot define default values for plot objects.

How MATLAB Software Searches for Default Values

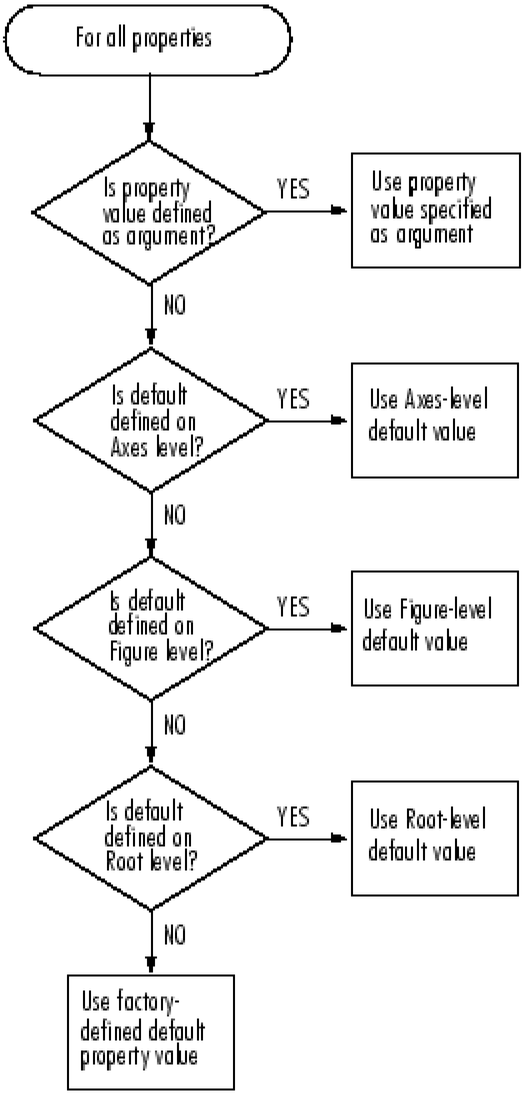
MATLAB searches for a default value beginning with the current object and continuing through the object's ancestors until it finds a user-defined default value or until it reaches the factory-defined value. Therefore, a search for property values is always satisfied.

The closer to the root of the hierarchy you define the default, the broader its scope. If you specify a default value for line objects on the root level, MATLAB uses that value for all lines because the root is at the top of the hierarchy. If you specify a default value for line objects on the axes level, MATLAB uses that value for line objects drawn only in that axes.

If you define default values on more than one level, the value defined on the closest ancestor takes precedence because MATLAB terminates the search as soon as it finds a value.

Setting default values affects only those objects created after you set the default. Existing graphics objects are not affected.

This diagram shows the steps MATLAB follows in determining the value of a graphics object property.



Defining Default Values

To specify default values, create a string beginning with the word `Default`, followed by the object type, and finally, by the object property. For example, to specify a default value of 1.5 points for the line property `LineWidth` at the level of the current figure, use the statement

```
set(gcf,'DefaultLineLineWidth',1.5)
```

The string `DefaultLineLineWidth` identifies the property as a line property. To specify the figure color, use `DefaultFigureColor`. It is meaningful to specify a default figure color only on the root level.

```
set(0,'DefaultFigureColor','b')
```

Use `get` to determine what default values are currently set on any given object level:

```
get(gcf,'default')
```

returns all default values set on the current figure.

Setting Properties to the Default

Specifying a property value of `'default'` sets the property to the first encountered default value defined for that property. For example, these statements result in a green surface `EdgeColor`:

```
set(0,'DefaultSurfaceEdgeColor','k')
h = surface(peaks);
set(gcf,'DefaultSurfaceEdgeColor','g')
set(h,'EdgeColor','default')
```

Because a default value for surface `EdgeColor` exists on the figure level, MATLAB encounters this value first and uses it instead of the default `EdgeColor` defined on the root.

Removing Default Values

Specifying a property value of `'remove'` gets rid of user-defined default values. The statement

```
set(0,'DefaultSurfaceEdgeColor','remove')
```

removes the definition of the default Surface EdgeColor from the root.

Setting Properties to Factory-Defined Values

Specifying a property value of 'factory' sets the property to its factory-defined value. (The property descriptions provides access to the factory settings for properties having predefined sets of values.)

For example, these statements set the EdgeColor of surface h to black (its factory setting), regardless of what default values you have defined:

```
set(gcf,'DefaultSurfaceEdgeColor','g')
h = surface(peaks);
set(h,'EdgeColor','factory')
```

Reserved Words

Setting a property value to default, remove, or factory produces the effects described in the previous sections. To set a property to one of these words (e.g., a text or uicontrol String property set to the word Default), you must precede the word with the backslash character:

```
h = uicontrol('Style','edit','String','\Default');
```

Examples — Setting Default Line Styles

The plot function cycles through the colors defined by the axes ColorOrder property when displaying multiline plots. If you define more than one value for the axes LineStyleOrder property, MATLAB increments the line style after each cycle through the colors.

You can set default property values that cause the plot function to produce graphs using varying line styles, but not varying colors. This is useful when you are working on a monochrome display or printing on a black and white printer.

First Example

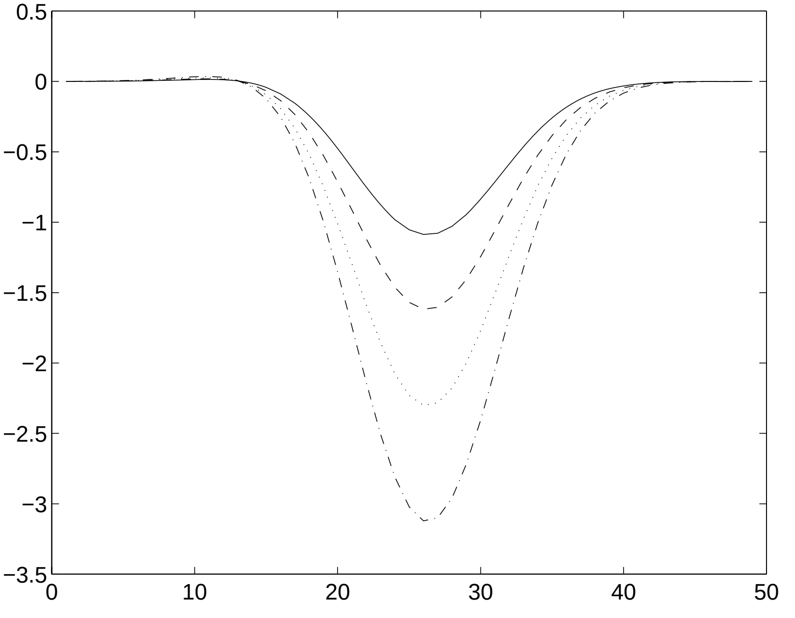
This example creates a figure with a white plot (axes) background color, and then sets default values for axes objects on the root level:

```
whitebg('w') %create a figure with a white color scheme
set(0,'DefaultAxesColorOrder',[0 0 0],...
      'DefaultAxesLineStyleOrder','-|--|:|-.')
```

Whenever you call `plot`,

```
Z = peaks; plot(1:49,Z(4:7,:))
```

it uses one color for all data plotted because the axes `ColorOrder` contains only one color, but it cycles through the line styles defined for `LineStyleOrder`.



Second Example

This example sets default values on more than one level in the hierarchy. These statements create two axes in one figure window, setting default values on the figure level and the axes level:

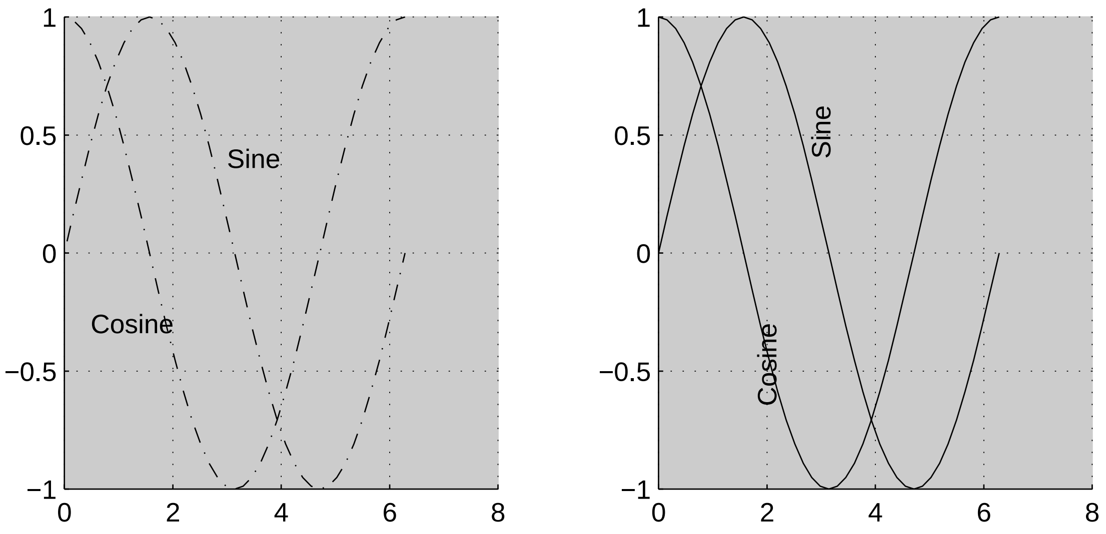
```
t = 0:pi/20:2*pi;
s = sin(t);
c = cos(t);
% Set default value for axes Color property
```

```
figh = figure('Position',[30 100 800 350],...
              'DefaultAxesColor',[.8 .8 .8]);

axh1 = subplot(1,2,1); grid on
% Set default value for line LineStyle property in first axes
set(axh1,'DefaultLineLineStyle','-.')
line('XData',t,'YData',s)
line('XData',t,'YData',c)
text('Position',[3 .4],'String','Sine')
text('Position',[2 -.3],'String','Cosine',...
     'HorizontalAlignment','right')

axh2 = subplot(1,2,2); grid on
% Set default value for text Rotation property in second axes
set(axh2,'DefaultTextRotation',90)
line('XData',t,'YData',s)
line('XData',t,'YData',c)
text('Position',[3 .4],'String','Sine')
text('Position',[2 -.3],'String','Cosine',...
     'HorizontalAlignment','right')
```

Issuing the same `line` and `text` statements to each subplot region results in a different display, reflecting different default settings.



Because the default axes `Color` property is set on the figure level of the hierarchy, MATLAB creates both axes with the specified gray background color.

The axes on the left (subplot region 121) defines a dash-dot line style (`-.`) as the default, so each call to the `line` function uses dash-dot lines. The axes on the right does not define a default line style, so MATLAB uses solid lines (the factory setting for lines).

The axes on the right defines a default text `Rotation` of 90 degrees, which rotates all text by this amount. MATLAB obtains all other property values from their factory settings, which results in nonrotated text on the left.

To install default values whenever you run MATLAB, specify them in your `startup.m` file. MATLAB might install default values for some appearance properties when started by calling the `colordef` command.

Accessing Object Handles

In this section...

Introduction

MATLAB software assigns a handle to every graphics object it creates. All object creation functions optionally return the handle of the created object. If you want to access the object's properties (e.g., from an M-file), assign its handle to a variable at creation time to avoid searching for it later.

You can always obtain the handle of an existing object with the `findobj` function or by listing its parent's `Children` property.

See "Searching for Objects by Property Values — findobj" on page 8-61 for examples.

See "Protecting Figures and Axes" on page 8-78 for more information on how object handles are hidden from normal access.

Special Object Handles

The root object's handle is always zero. The handle of a figure is either:

- An integer that, by default, is displayed in the window title bar
- A floating point number requiring full MATLAB internal precision

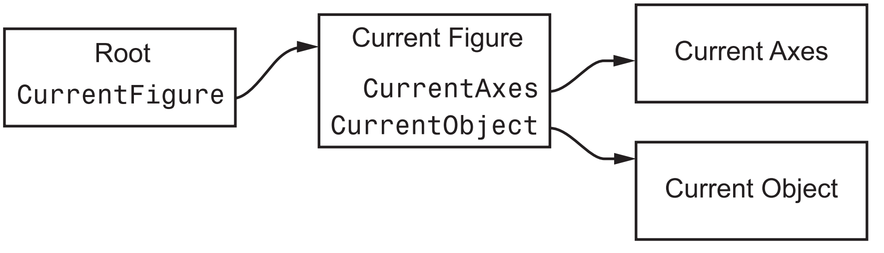
The figure `IntegerHandle` property controls the type of handle the figure receives.

All other graphics object handles are floating-point numbers. You must maintain the full precision of these numbers when you reference handles. Rather than attempting to read handles off the screen and retype them, you must store the value in a variable and pass that variable whenever MATLAB requires a handle.

The Current Figure, Axes, and Object

An important concept in the Handle Graphics technology is that of being current. The current figure is the window designated to receive graphics output. Likewise, the current axes is the target for commands that create axes children. The current object is the last graphics object created or clicked on by the mouse.

MATLAB stores the three handles corresponding to these objects in the ancestor's property list.



These properties enable you to obtain the handles of these key objects:

```
get(0,'CurrentFigure');
get(gcf,'CurrentAxes');
get(gcf,'CurrentObject');
```

The following commands are shorthand notation for the `get` statements:

- `gcf` — Returns the value of the root `CurrentFigure` property.
- `gca` — Returns the value of the current figure's `CurrentAxes` property.
- `gco` — Returns the value of the current figure's `CurrentObject` property.

Use these commands as input arguments to functions that require object handles. For example, you can click a line object and then use `gco` to specify the handle to the `set` command,

```
set(gco,'Marker','square')
```

or list the values of all current axes properties with

```
get(gca)
```

You can get the handles of all the graphic objects in the current axes (except those with hidden handles),

```
h = get(gca,'Children');
```

and then determine the types of the objects.

```
get(h,'type')
ans =
       'text'
       'patch'
       'surface'
       'line'
```

While `gcf` and `gca` provide a simple means of obtaining the current figure and axes handles, they are less useful in M-files. This is particularly true if your M-file is part of an application layered on MATLAB where you do not necessarily have knowledge of user actions that can change these values.

See "Controlling Graphics Output" on page 8-70 for information on how to prevent users from accessing the handles of graphics objects that you want to protect.

Searching for Objects by Property Values — findobj

The `findobj` function provides a means to traverse the object hierarchy quickly and obtain the handles of objects having specific property values. To serve as a means of identification, all graphics objects have a `Tag` property that you can set to any string. You can then search for the specific property/value pair.

For example, suppose you create a checkbox that is sometimes inactivated in the GUI. By assigning a unique value for the `Tag` property, you can always find that particular instance and set its properties:

```
uicontrol('Style','checkbox','Tag','save option')
```

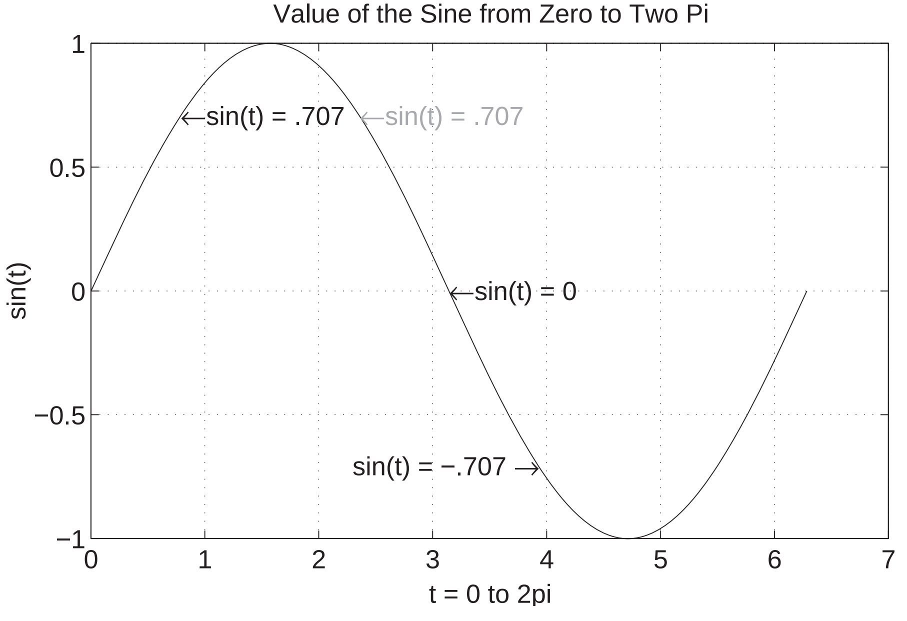
Use `findobj` to locate the object whose `Tag` property is set to `'save option`' and disable it:

```
set(findobj('Tag','save option'),'Enable','off')
```

If you do not specify a starting object, `findobj` searches from the root object, finding all occurrences of the property name/property value combination that you specify.

Example — Finding Objects

This plot of the sine function contains text objects labeling particular values of the function.



Suppose you want to move the text string labeling the value sin(t) = .707 from its current location at `[pi/4,sin(pi/4)]`, to the point `[3*pi/4,sin(3*pi/4)]` where the function has the same value (shown grayed out in the picture). To do this, determine the handle of the text object labeling that point and change its `Position` property.

To use `findobj`, pick a property value that uniquely identifies the object. This example uses the text `String` property:

```
text_handle = findobj('String','\leftarrowsin(t) = .707');
```

Move the object to the new position, defining the text `Position` in axes units.

```
set(text_handle,'Position',[3*pi/4,sin(3*pi/4),0])
```

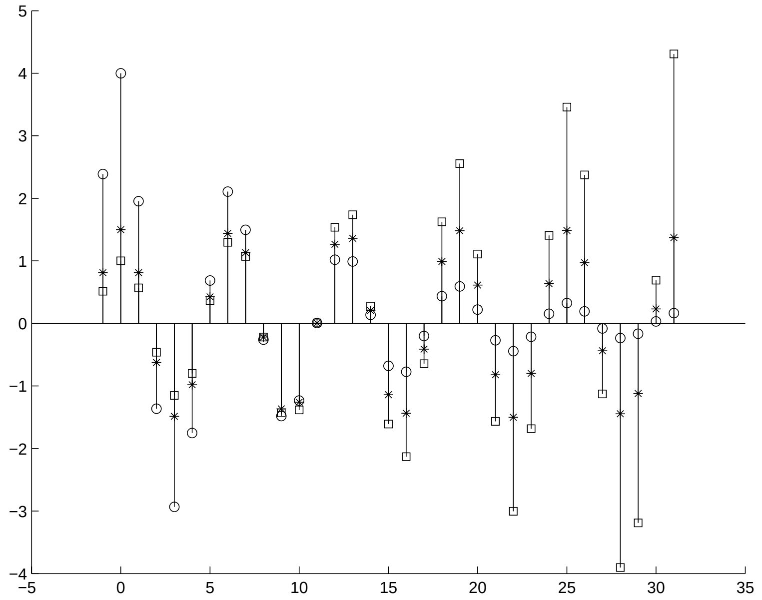
`findobj` lets you restrict the search by specifying a starting point in the hierarchy, instead of beginning with the root object. This results in faster searches if there are many objects in the hierarchy. In the previous example, you know the text object of interest is in the current axes, so you can type

```
text_handle = findobj(gca,'String','\leftarrowsin(t) = .707');
```

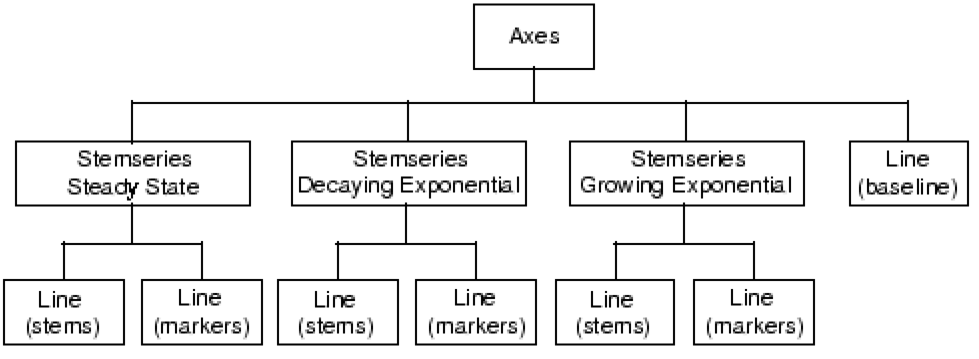
Example — Using Logical Operators and Regular Expression

Suppose you create the following graph and want to modify certain properties of the objects created:

```
x = 0:30;
y = [1.5*cos(x);4*exp(-.1*x).*cos(x);exp(.05*x).*cos(x)]';
h = stem(x,y);
set(h(1),'Color','black',...
            'Marker','o',...
            'Tag','Decaying Exponential')
set(h(2),'Color','black',...
            'Marker','square',...
            'Tag','Growing Exponential')
set(h(3),'Color','black',...
            'Marker','*',...
            'Tag','Steady State')
```



The following instance diagram shows the graphics objects created in the graph. Each of the three sets of data produces a stemseries object, which in turn uses two lines to create the stem graph: one line for the stems and one for the markers that terminate each stem. There is also a line used for the baseline.



Controlling the Depth of the Search. Make the baseline into a dashed line. Because it is parented directly to the axes, use the following statement to access only this line:

```
set(findobj(gca,'-depth',1,'Type','line'),'LineStyle','--')
```

By setting `-depth` to `1`, `findobj` searches only the axes and its immediate children. As you can see from the above instance diagram, the baseline is the only line object parented directly to the axes.

Limiting the Search with Regular Expressions. Increase the value of the `MarkerSize` property by 2 points on all stemseries objects that do not have their property `Tag` set to `'Steady State'`:

```
h = findobj('-regexp','Tag','^(?!Steady State$).');
set(h,{'MarkerSize'},num2cell(cell2mat(get(h,'MarkerSize'))+2))
```

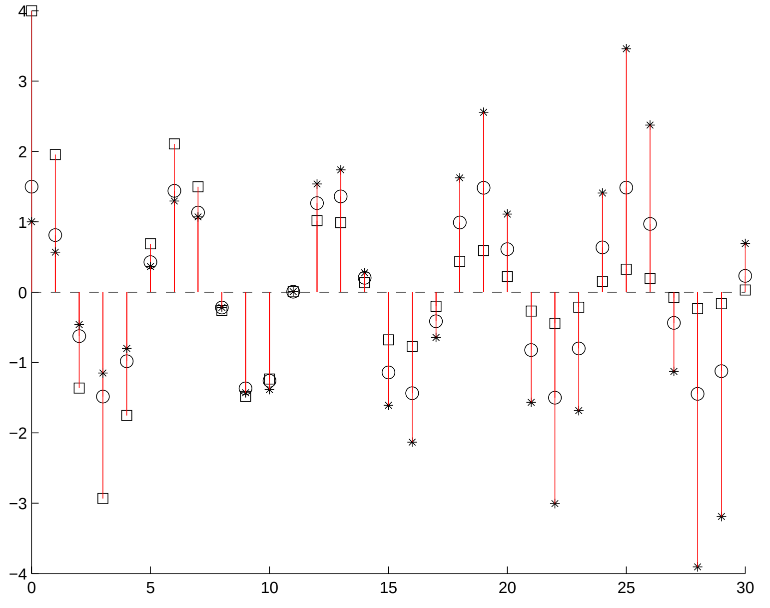
See the `regexp` function for more information on using regular expressions in MATLAB.

Using Logical Operators. Change the color of the stem lines, but not the stem markers. To do this, you must access the line objects contained by the three stemseries objects. You cannot just set the stemseries `Color` property because it sets both the line and marker colors.

Search for objects that are of `Type` `line`, have `Marker` set to `none`, and do not have `LineStyle` set to `'--'`, which is the baseline:

```
h = findobj('type','line','Marker','none',...
    '-and','-not','LineStyle','--');
set(h,'Color','red')
```

The following picture shows the graph after making these changes.



Copying Objects

You can copy objects from one parent to another using the `copyobj` function. The new object differs from the original object only in the value of its `Parent` property and its handle; otherwise it is a clone of the original. You can copy a number of objects to a new parent, or one object to a number of new parents, as long as the result maintains the correct parent/child relationship.

When you copy an object having child objects, MATLAB copies all children as well.

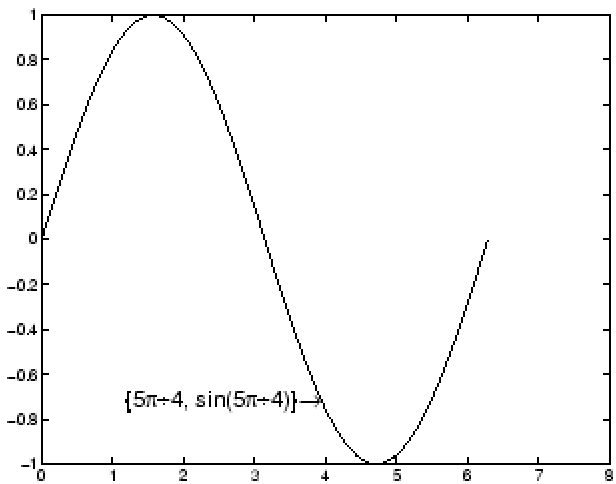
Example — Copying Objects

Suppose you are plotting a variety of data and want to label the point having the *x*- and *y*-coordinates determined by $5\pi \div 4$, $\sin(5\pi \div 4)$ in each plot. The `text` function allows you to specify the location of the label in the coordinates defined by the *x*- and *y*-axis limits, simplifying the process of locating the text:

```
text('String','\{5\pi\div4, sin(5\pi\div4)\}\rightarrow',...
     'Position',[5*pi/4,sin(5*pi/4),0],...
     'HorizontalAlignment','right')
```

In this statement, the `text` function:

- Labels the data point with the string $\{5\pi \div 4, \sin(5\pi \div 4)\}$ using TeX commands to draw a right-facing arrow and mathematical symbols.
- Specifies the `Position` in terms of the data being plotted.
- Places the data point to the right of the text string by changing the `HorizontalAlignment` to `right` (the default is `left`).

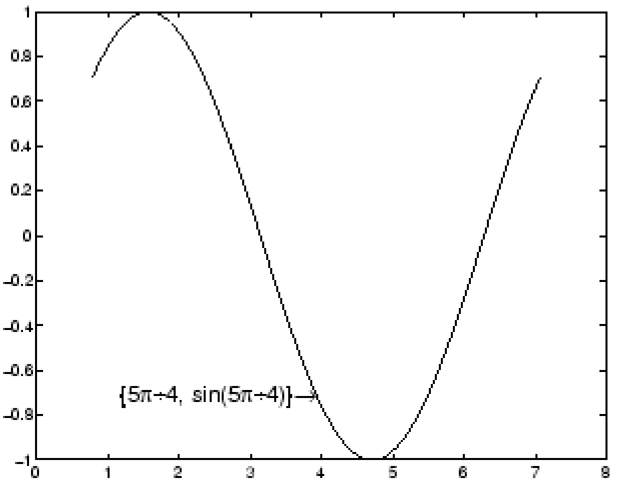


To label the same point with the same string in another plot, copy the text using `copyobj`. Because the last statement did not save the handle to the text object, you can find it using `findobj` and the `'String'` property:

```
text_handle = findobj('String',...
              '\{5\pi\div4,sin(5\pi\div4)\}\rightarrow');
```

After creating the next plot, add the label by copying it from the first plot:

```
copyobj(text_handle,gca).
```



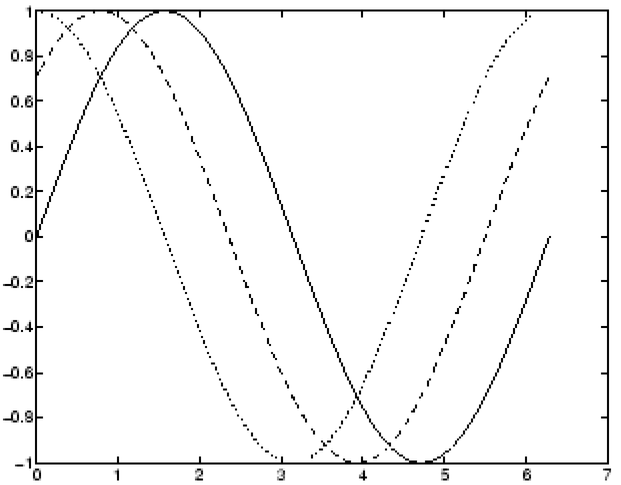
This particular example takes advantage of the fact that text objects define their location in the axes data space. Therefore, the text `Position` property did not need to change from one plot to another.

Deleting Objects

You can remove a graphics object with the `delete` command, using the object's handle as an argument. For example, you can delete the current axes (and all of its descendants) with the statement

```
delete(gca)
```

You can use `findobj` to get the handle of a particular object you want to delete. For example, to find the handle of the dotted line in this multiline plot,



use `findobj` to locate the object whose `LineStyle` property is `':'`

```
line_handle = findobj('LineStyle',':');
```

Use this handle with the `delete` command:

```
delete(line_handle)
```

You can combine these two statements, substituting the `findobj` statement for the handle:

```
delete(findobj('LineStyle',':'))
```

Controlling Graphics Output

In this section...

Figure Targets

MATLAB software allows many figure windows to be open simultaneously during a session. A MATLAB application might create figures to display graphical user interfaces and plotted data. You need to protect some figures from becoming the target for graphics display and to prepare (e.g., reset properties and clear existing objects from) others before receiving new graphics.

Specifying the Target for Graphics Output

By default, MATLAB functions that create graphics objects display them in the current figure and current axes (if it is an axes child). You can direct the output to another parent by explicitly specifying the `Parent` property with the creating function. For example,

```
plot(1:10,'Parent',axes_handle)
```

where `axes_handle` is the handle of the target axes. The `uicontrol` and `uimenu` functions have a convenient syntax that enables you to specify the parent as the first argument,

```
uicontrol(figure_handle,...)
uimenu(parent_menu_handle,...)
```

or you can set the `Parent` property. Many plotting functions accept an axes handle as the first argument as well.

Making a Figure and Axes Current

You can specify which figure and which axes within the figure are the target for graphics output. There are two ways to do this.

Making Current and Update. If `figure_handle` is the handle to an existing figure, then the statement

```
figure(figure_handle)
```

- Makes `figure_handle` the current figure.
- Restacks `figure_handle` to be the front-most figure displayed.
- Makes `figure_handle` visible if it was not.
- Refreshes `figure_handle` and process all pending window events.

The same behavior applies to axes. The statement

```
axes(axes_handle)
```

- Makes `axes_handle` the current axes.
- Restacks `axes_handle` to be the front-most axes displayed.
- Makes `axes_handle` visible if it was not.
- Refreshes the figure containing the axes and process all pending window events for that figure.

Make Current Without Changing State. You can make a figure or axes current without causing MATLAB to change the object's state by setting the figure's root object `CurrentFigure` property or the figure object's `CurrentAxes` property to the handle of the figure or axes you want to accept graphics output.

If `figure_handle` is the handle to an existing figure, the statement

```
set(0,'CurrentFigure',figure_handle)
```

makes `figure_handle` the current figure without changes its state. Similarly, if `axes_handle` is the handle of an axes object, the statement

```
set(h,'CurrentAxes',axes_handle)
```

makes it the current axes, assuming `h` is the handle of the figure that contains it.

Preparing Figures and Axes for Graphics

By default, commands that generate graphics output display the graphics objects in the current figure without clearing or resetting figure properties. However, if the graphics objects are axes children, MATLAB clears the axes and resets most axes properties to their default values before displaying the objects.

You can change this behavior by setting the figure and axes `NextPlot` properties.

Using NextPlot to Control Output Target

MATLAB high-level graphics functions check the values of the `NextPlot` properties to determine whether to add, clear, or clear and reset the figure and axes before drawing. Low-level object-creation functions do not check the `NextPlot` properties. They simply add the new graphics objects to the current figure and axes.

Low-level functions are designed primarily for use in M-files where you can implement whatever drawing behavior you want. However, when you develop a MATLAB-based application, controlling MATLAB drawing behavior is essential to creating a program that behaves predictably.

This table summarizes the possible values for the `NextPlot` property.

NextPlot	Figure	Axes
`new`	Create a new figure and use it as the current figure.	Not an option for axes.

NextPlot	Figure	Axes
`add`	Add new graphics objects without clearing or resetting the current figure. (Default)	Add new graphics objects without clearing or resetting the current axes.
`replacechildren`	Remove all child objects, but do not reset figure properties. Equivalent to `clf`.	Remove all child objects, but do not reset axes properties. Equivalent to `cla`.
`replace`	Remove all child objects and reset figure properties to their defaults. Equivalent to `clf reset`.	Remove all child objects and reset axes properties to their defaults. Equivalent to `cla reset`. (Default)

A `reset` returns all properties, except `Position` and `Units`, to their default values.

The `hold` command provides convenient access to the `NextPlot` properties. The statement

```
hold on
```

sets both figure and axes `NextPlot` properties to `add`.

The statement

```
hold off
```

sets the axes `NextPlot` property to `replace`.

Targeting Graphics Output with newplot

MATLAB provides the `newplot` function to simplify the process of writing graphics M-files that conform to the settings of the `NextPlot` properties.

newplot checks the values of the NextPlot properties and takes the appropriate action based on these values. Place newplot at the beginning of any M-file that calls object creation functions.

When your M-file calls newplot, the following possible actions occur:

1. newplot checks the current figure's NextPlot property:
   - If there are no figures in existence, newplot creates one and makes it the current figure.
   - If the value of NextPlot is add, newplot makes the figure the current figure.
   - If the value of NextPlot is new, newplot creates a new figure and makes it the current figure
   - If the value of NextPlot is replacechildren, newplot deletes the figure's children (axes objects and their descendants) and makes this figure the current figure.
   - If the value of NextPlot is replace, newplot deletes the figure's children, resets the figure's properties to the defaults, and makes this figure the current figure.
2. newplot checks the current axes' NextPlot property:
   - If there are no axes in existence, newplot creates one and makes it the current axes.
   - If the value of NextPlot is add, newplot makes the axes the current axes.
   - If the value of NextPlot is replacechildren, newplot deletes the axes' children and makes this axes the current axes.
   - If the value of NextPlot is replace, newplot deletes the axes' children, resets the axes' properties to the defaults, and makes this axes the current axes.

MATLAB Default Behavior

Consider the default situation where the figure NextPlot property is add and the axes NextPlot property is replace. When you call newplot, it:

1 Checks the value of the current figure's NextPlot property (which is add) and determines MATLAB can draw into the current figure with no further action. If there is no current figure, newplot creates one, but does not recheck its NextPlot property.

2 Checks the value of the current axes' NextPlot property (which is replace), deletes all graphics objects from the axes, resets all axes properties (except Position and Units) to their defaults, and returns the handle of the current axes.

Example — Using newplot

To illustrate the use of newplot, this example creates a function similar to the plot function, except it automatically cycles through different line styles instead of using different colors for multiline plots.

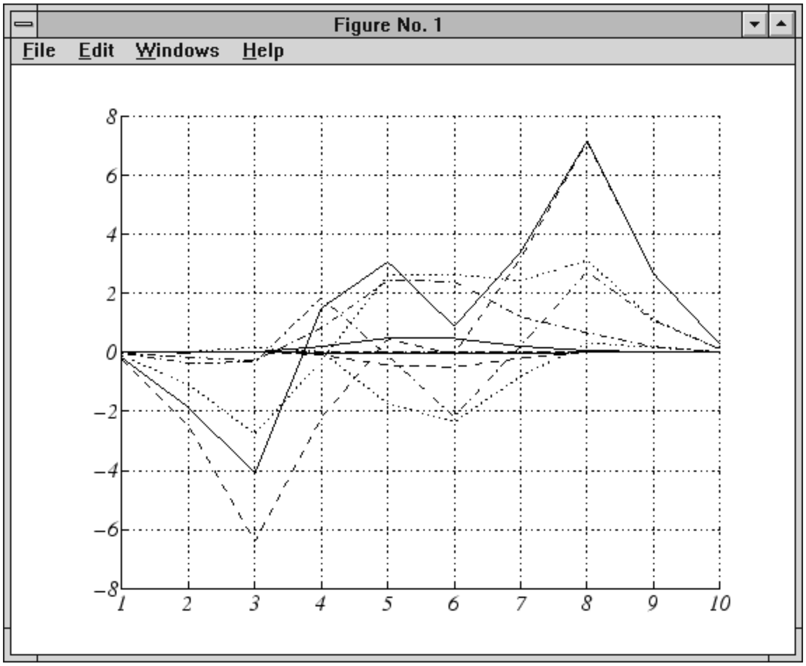
```
function my_plot(x,y)
cax = newplot;   % newplot returns handle of current axes
LSO = ['- ';'--';': ';'-.'];
set(cax,'FontName','Times','FontAngle','italic')
set(get(cax,'Parent'),'MenuBar','none')
line_handles = line(x,y,'Color','b');
style = 1;
for i = 1:length(line_handles)
    if style > length(LSO), style = 1;end
    set(line_handles(i),'LineStyle',LSO(style,:))
    style = style + 1;
end
grid on
```

The my_plot function uses the high-level line function syntax to plot the data. This provides the same flexibility in input argument dimension that the plot function supports. The line function does not check the value of the figure or axes NextPlot property. However, because my_plot calls newplot, it behaves the same way the high-level plot function does—with default values in place, my_plot clears and resets the axes each time you call it.

my_plot uses the handle returned by newplot to access the target figure and axes. This example sets axes font properties and disables the figure's menu bar. You obtain the figure handle via the axes Parent property.

This picture shows typical output for the `my_plot` function.

```
my_plot(1:10,peaks(10))
```



Basic Plotting M-File Structure

This example:

- Calls `newplot` early to conform to the `NextPlot` properties and to obtain the handle of the target axes.
- References the axes handle returned by `newplot` to set any axes properties or to obtain the figure's handle.
- Calls object creation functions to draw graphics objects with the desired characteristics.

The MATLAB default settings for the `NextPlot` properties facilitate writing M-files that adhere to the standard behavior: reuse the figure window,

but clear and reset the axes with each new graph. Other values for these properties allow you to implement different behaviors.

Replacing Only the Child Objects — replacechildren

The `replacechildren` value for `NextPlot` causes `newplot` to remove child objects from the figure or axes, but does not reset any property values (except the list of handles contained in the `Children` property).

This can be useful after setting properties you want to use for subsequent graphs without having to reset properties. For example, if you type at the command line

```
set(gca,'ColorOrder',[0 0 1],'LineStyleOrder','-|--|:|-.',...
        'NextPlot','replacechildren')
plot(x,y)
```

`plot` produces the same output as the `my_plot` M-file in the previous section, but only within the current axes. Calling `plot` still erases the existing graph (i.e., deletes the axes children), but it does not reset axes properties. The values specified for the `ColorOrder` and `LineStyleOrder` properties remain in effect.

Testing for Hold State

In some situations your M-file should change the visual appearance of the axes to accommodate new graphics objects. For example, if you want the `my_plot` M-file from the previous example to accept 3-D data, it makes sense to set the view to 3-D when the input data has *z*-coordinates.

However, to be consistent with the behavior of the MATLAB high-level routines, it is good practice to test whether `hold` is `on` before changing parent axes or figure properties. When `hold` is `on`, the axes and figure `NextPlot` properties are both set to `add`.

The M-file `my_plot3` accepts 3-D data and also checks the hold state, using `ishold`, to determine whether it should change the view.

```
function my_plot3(x,y,z)
cax = newplot;
hold_state = ishold; % ishold tests the current hold state
```

```
LSO = ['- ';'--';': ';'-.'];
if nargin == 2
    hlines = line(x,y,'Color','k');
    if ~hold_state % Change view only if hold is off
        view(2)
    end
elseif nargin == 3
    hlines = line(x,y,z,'Color','k');
    if ~hold_state % Change view only if hold is off
        view(3)
    end
end
ls = 1;
for hindex = 1:length(hlines)
    if ls > length(LSO),ls = 1;end
    set(hlines(hindex),'LineStyle',LSO(ls,:))
    ls = ls + 1;
end
```

If `hold` is `on` when you call `my_plot3`, it does not change the view. If `hold` is `off`, `my_plot3` sets the view to 2-D or 3-D, depending on whether there are two or three input arguments.

Protecting Figures and Axes

In some situations it is important to prevent particular figures or axes from becoming the target for graphics output (i.e., preventing them from becoming the `gcf` or `gca`). An example is a figure containing the uicontrols that implement a user interface.

You can prevent MATLAB from drawing into a particular figure or axes by removing its handle from the list of handles that are visible to the `newplot` function, as well as any other functions that either return or implicitly reference handles (i.e., `gca`, `gcf`, `gco`, `cla`, `clf`, `close`, and `findobj`). Two properties control handle hiding: `HandleVisibility` and `ShowHiddenHandles`.

HandleVisibility Property

HandleVisibility is a property of all objects. It controls the scope of handle visibility within three different ranges. Property values can be:

- on — The object's handle is available to any function executed on the MATLAB command line or from an M-file. This is the default.
- callback — The object's handle is hidden from all functions executing at the command line, even if it is on the top of the screen stacking order. However, during callback routine execution (MATLAB statements or functions that execute in response to user action), the handle is visible to all functions, such as gca, gcf, gco, findobj, and newplot. This setting enables callback routines to take advantage of the MATLAB handle access functions, while ensuring that users typing at the command line do not inadvertently disturb a protected object.
- off — The object's handle is hidden from all functions executing on the command line and in callback routines. This setting is useful when you want to protect objects from possibly damaging user commands.

For example, if a GUI accepts user input in the form of text strings, which are then evaluated (using the eval function) from within the callback routine, a string such as 'close all' could destroy the GUI. To protect against this situation, you can temporarily set HandleVisibility to off on key objects:

```
user_input = get(editbox_handle,'String');
set(gui_handles,'HandleVisibility','off')
eval(user_input)
set(gui_handles,'HandleVisibility','on')
```

Values Returned by gca and gcf. When a protected figure is topmost on the screen, but has unprotected figures stacked beneath it, gcf returns the topmost unprotected figure in the stack. The same is true for gca. If no unprotected figures or axes exist, calling gcf or gca causes MATLAB to create one in order to return its handle.

Accessing Protected Objects

The root ShowHiddenHandles property enables and disables handle visibility control. By default, ShowHiddenHandles is off, which means MATLAB obeys the setting of the HandleVisibility property. When ShowHiddenHandles is

set to `on`, all handles are visible from the command line and within callback routines. This can be useful when you want access to all graphics objects that exist at a given time, including the handles of axes text labels, which are normally hidden.

The `close` function also allows access to nonvisible figures using the `hidden` option. For example,

```
close('hidden')
```

closes the topmost figure on the screen, even if it is protected. Combining `all` and `hidden` options,

```
close('all','hidden')
```

closes all figures.

Handle Validity Versus Handle Visibility

All handles remain valid regardless of whether they are visible or not. If you know an object's handle, you can set and get its properties. By default, figure handles are integers that are displayed at the top of the window.

You can provide further protection to figures by setting the `IntegerHandle` property to `off`. MATLAB then uses a floating-point number for figure handles.

The Figure Close Request Function

In this section...

Introduction

MATLAB software executes a callback routine defined by the figure's `CloseRequestFcn` whenever you:

- Issue a `close` command on a figure.
- Quit MATLAB while there are visible figures. (If a figure's `Visible` property is set to `off`, MATLAB does not execute its close request function when you quit MATLAB; the figure is just deleted.)
- Close a figure from the windowing system using a close box or a close menu item.

The close request function enables you to prevent or delay the closing of a figure or the termination of a MATLAB session. This is useful to perform such actions as

- Displaying a dialog box requiring the user to confirm the action.
- Saving data before closing.
- Preventing unintentional command-line deletion of a graphical user interface built with MATLAB.

The default callback routine for the `CloseRequestFcn` is an M-file called `closereq`. It contains the statements

```
if isempty(gcbf)
   if length(dbstack) == 1
      warning('MATLAB:closereq',...
      'Calling closereq from the command line is now obsolete,...
```

```
       use close instead');
      end
      close force
   else
      delete(gcbf);
   end
```

This callback honors `HandleVisibility` and therefore does not delete the figure when you use the `close` command without specifying the figure handle. For example:

```
h = figure('HandleVisibility','off')
close     % figure does not close
close all % figure does not close
close(h)  % figure closes
```

Quitting the MATLAB Environment

When you quit MATLAB, the current figure's `CloseRequestFcn` is called, and if the figure is deleted, the next figure in the root's list of children (i.e., the root's `Children` property) becomes the current figure, and its `CloseRequestFcn` is in turn executed, and so on. You can use `gcbf` to specify the figure handle from within a user-written close request function.

If you change a figure's `CloseRequestFcn` so that it does not delete the figure, issuing the close command on that figure does not cause it to be deleted. Furthermore, if you attempt to quit MATLAB, the quit is aborted because MATLAB does not delete the figure.

Errors in the Close Request Function

If the `CloseRequestFcn` generates an error when executed, MATLAB aborts the close operation. However, errors in the `CloseRequestFcn` do not abort attempts to quit MATLAB. If an error occurs in a figure's `CloseRequestFcn`, MATLAB closes the figure unconditionally following a `quit` or `exit` command.

Overriding the Close Request Function

The `delete` command always deletes the specified figure, regardless of the value of its `CloseRequestFcn`. For example, the statement

```
delete(get(0,'Children'))
```

deletes all figures whose handles are not hidden (i.e., the figures' `HandleVisibility` property is not set to `off`). If you want to delete all figures regardless of whether their handles are hidden, you can set the root `ShowHiddenHandles` property to `on`. The root `Children` property then contains the handles of all figures. For example, the statements

```
set(0,'ShowHiddenHandles','yes')
delete(get(0,'Children'))
```

unconditionally delete all figures.

Saving Handles in M-Files

In this section...

About Saving Handles

Graphics M-files frequently use handles to access property values and to direct graphics output to a particular target. MATLAB software provides utility routines that return the handles to key objects (such as the current figure and axes). In M-files, however, these utilities might not be the best way to obtain handles because:

- Querying MATLAB for the handle of an object or other information is less efficient than storing the handle in a variable and referencing that variable.
- The current figure, axes, or object might change during M-file execution because of user interaction.

Save Information First

It is good practice to save relevant information about the MATLAB state in the beginning of your M-file. For example, you can begin an M-file with

```
cax = newplot;
cfig = get(cax,'Parent');
hold_state = ishold;
```

rather than querying this information each time you need it. Remember that utility commands like `ishold` obtain the values they return whenever called. (The `ishold` command issues a number of `get` commands and string compares (`strcmp`) to determine the hold state.)

If you are temporarily going to alter the hold state within the M-file, save the current values of the `NextPlot` properties so you can reset them later:

```
ax_nextplot = lower(get(cax,'NextPlot'));
fig_nextplot = lower(get(cfig,'NextPlot'));
.
```

```
.
.
set(cax,'NextPlot',ax_nextplot)
set(cfig,'NextPlot',fig_nextplot)
```

Properties Changed by Built-In Functions

To achieve their intended effect, many built-in functions change axes properties, which can then affect the workings of your M-file. This table lists the MATLAB built-in graphics functions and the properties they change. These properties change only if `hold` is `off`.

Function	Axes Property: Set To
`fill`	`Box: on` `CameraPosition`: 2-D view `CameraTarget`: 2-D view `CameraUpVector`: 2-D view `CameraViewAngle`: 2-D view

`fill3`	`CameraPosition`: 3-D view
	`CameraTarget`: 3-D view
	`CameraUpVector`: 3-D view
	`CameraViewAngle`: 3-D view
	`XScale: linear`
	`YScale: linear`
	`ZScale: linear`

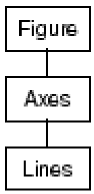
`image` (high-level)	`Box: on`
	`Layer: top`
	`CameraPosition`: 2-D view
	`CameraTarget`: 2-D view
	`CameraUpVector`: 2-D view
	`CameraViewAngle`: 2-D view
	`XDir: normal`
	`XLim: [0 size(CData,2)]+0.5`
	`XLimMode: manual`

Function	Axes Property: Set To
	YDir: reverse YLim: [0 size(CData,1)]+0.5YLimMode: manual
loglog	Box: on CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view XScale: log YScale: log
plot	Box: on CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view
plot3	CameraPosition: 3-D view CameraTarget: 3-D view CameraUpVector: 3-D view CameraViewAngle: 3-D view XScale: linear YScale: linear ZScale: linear

Function	Axes Property: Set To
semilogx	Box: on CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view XScale: log YScale: linear
semilogy	Box: on CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view XScale: linear YScale: log

Objects That Can Contain Other Objects

Certain graphics objects can contain other objects. Consider a graph, for example. In a graph, an object like a line represents data . Normally, the parent of the line is an axes (i.e., the handle of the line's `Parent` property is set to the handle of the axes that contains it). A figure is normally the parent of an axes. A typical object diagram of a graph looks like the next figure.



When graphs become more complicated and represent data with multiple objects, you can group these objects together so you can perform operations on the group as a whole.

The following sections discuss how to use two container objects that group axes children within a graph and user interface components within a figure.

Using Panel Containers in Figures — Uipanels

In this section...

Introduction

Figures can contain axes and user interface objects directly, or you can parent these objects to uipanels, which you then parent to a figure. Uipanels are useful for the design of GUIs because they enable you to define subregions in a figure in which you can lay out components.

MATLAB software interprets the `Position` property of all objects parented to a uipanel relative to the uipanel's position. If you move the uipanel, the children automatically move with it.

Uipanels can also contain other uipanels, as well as axes, uicontrols, and uibuttongroups. See the `uipanel` reference page for more information on uipanels.

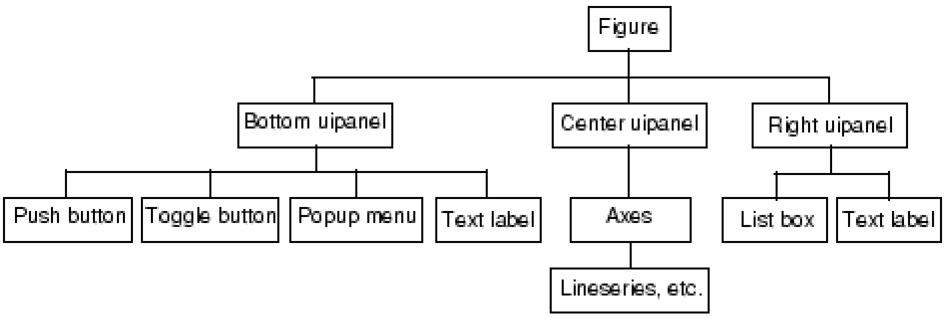
You can create multiple axes in a uipanel and direct plotting into any of them. However, some plotting functions do not allow you to specify the parent of the graphics objects they create, so they create a new axes (and possibly a figure). To include such a graph in a uipanel, reparent the axes to the panel once the plot is made.

Figure Resize Functions

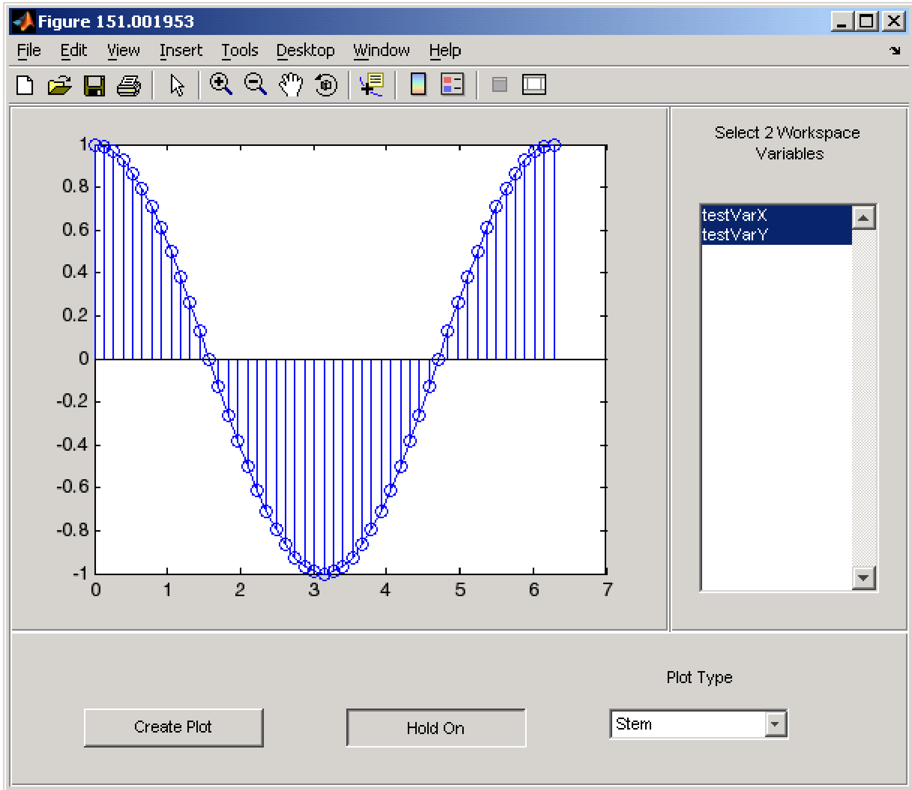
Containing various parts of a GUI in uipanels simplifies the process of programming figure resize behavior because you can write a separate resize function for each panel. The following example illustrates how to do this.

Example — Using Figure Panels

This example uses three uipanel objects as containers for the GUI's components. All three uipanels are then parented to the figure, as shown in the following containment hierarchy.



Here is a GUI with some data plotted in the axes.



Complete Example Code

Note If you are using the MATLAB Help browser, run this example or open it in the MATLAB editor.

This GUI enables you to select workspace variables from a list box and select a plot type from a pop-up menu. You can add plots to the existing graph by clicking the **Hold** toggle button and initiate the plot by clicking the **Create Plot** button.

Use the link above to run the example and open the GUI code in the MATLAB editor.

Creating the Uipanels

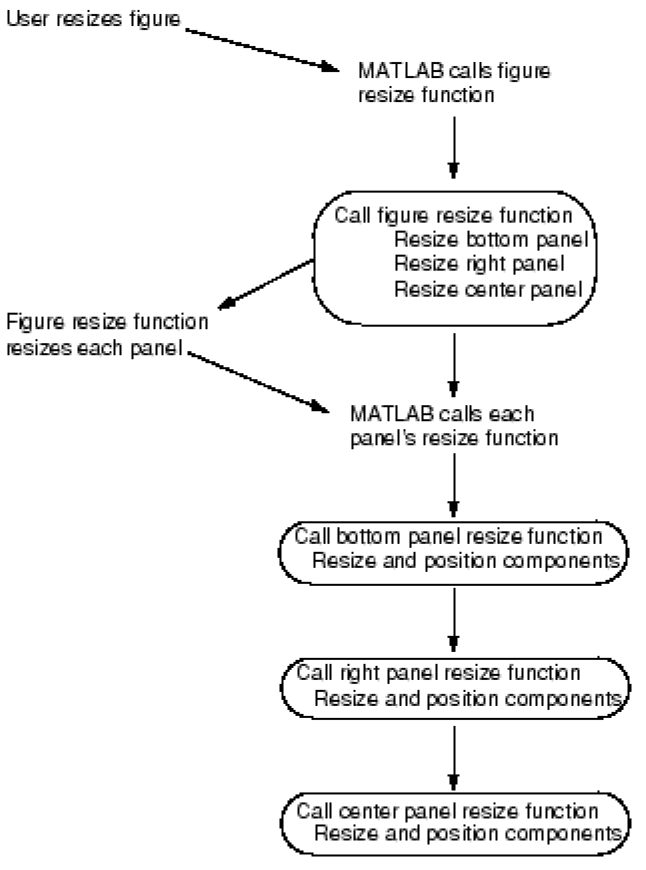
The following code shows the definition of the figure and the bottom panel. Setting `Units` to `characters` ensures that your GUI is properly sized on different computer systems. The `Position` property specifies the location and size of each component in units set by the `Units` property.

```
% Create the figure
f = figure('Units','characters',...
   'Position',[30 30 120 35],...
   'Color',panelColor,...
   'HandleVisibility','callback',...
   'IntegerHandle','off',...
   'Renderer','painters',...
   'ResizeFcn',@figResize);
% Create the bottom uipanel
botPanel = uipanel('BorderType','etchedin',...
   'BackgroundColor',panelColor,...
   'Units','characters',...
   'Position',[1/20 1/20 119.9 8],...
   'Parent',f,...
   'ResizeFcn',@botPanelResize);
```

Programming the Resize Functions

As you resize the figure, MATLAB calls the figure resize function (specified by the object's `ResizeFcn` property), which, in this example, computes a new size for each uipanel. Because the figure resize function resizes the uipanels, MATLAB automatically calls the resize function of each uipanel once the figure resize function completes execution. The uipanel resize functions then adjust the sizes and locations of the components they contain.

The following diagram illustrates the sequence of events that occurs when a user resizes the figure.



The following code shows the figure, bottom panel, and right panel resize functions. As each function is called, it sets the object's size and position to values that are proportional to the original layout.

See "Nested Functions" for more information.

```
% Figure resize function
function figResize(src,evt)
 fpos = get(f,'Position');
```

```
 set(botPanel,'Position',...
  [1/20 1/20 fpos(3)-.1 fpos(4)*8/35])
 set(rightPanel,'Position',...
   [fpos(3)*85/120 fpos(4)*8/35 fpos(3)*35/120 fpos(4)*27/35])
 set(centerPanel,'Position',...
  [1/20 fpos(4)*8/35 fpos(3)*85/120 fpos(4)*27/35]);
end
% Bottom panel resize function
function botPanelResize(src,evt)
 bpos = get(botPanel,'Position');
 set(plotButton,'Position',...
  [bpos(3)*10/120 bpos(4)*2/8 bpos(3)*24/120 2])
 set(holdToggle,'Position',...
  [bpos(3)*45/120 bpos(4)*2/8 bpos(3)*24/120 2])
 set(popUp,'Position',...
  [bpos(3)*80/120 bpos(4)*2/8 bpos(3)*24/120 2])
 set(popUpLabel,'Position',...
  [bpos(3)*80/120 bpos(4)*4/8 bpos(3)*24/120 2])
end
% Right panel resize function
function rightPanelResize(src,evt)
 rpos = get(rightPanel,'Position');
 set(listBox,'Position',...
  [rpos(3)*4/32 rpos(4)*2/27 rpos(3)*24/32 rpos(4)*20/27]);
 set(listBoxLabel,'Position',...
  [rpos(3)*4/32 rpos(4)*24/27 rpos(3)*24/32 rpos(4)*2/27]);
end
```

The center panel does not need a resize function because the axes automatically resize to fit the container (either a figure or uipanel).

To see the complete code listing for this example, see "Complete Example Code" on page 8-92.

Grouping Objects Within Axes — hgtransform

In this section...

Introduction

MATLAB software provides two objects designed to group any of the objects normally parented to axes:

- Hggroup — Parent objects to an hggroup object when you want to reference the objects as a group, for example, to select or control visibility of all the group members.
- hgtransform — This object also enables you to transform (rotate, translate, etc.) the objects as a group.

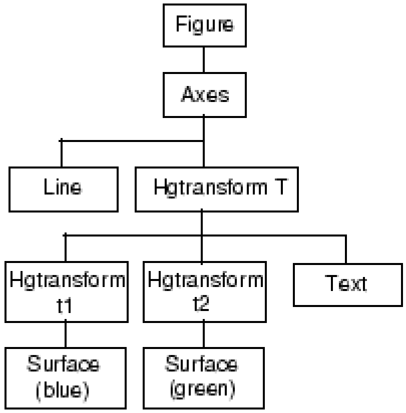
See "Group Objects" on page 8-28 for more information about hggroup and hgtransform objects.

Example — Translating Grouped Objects

This example shows how using a hierarchy of hgtransform objects makes it possible to translate the contained graphics objects both independently and as a group. The example creates a cross-like cursor with a text readout in the center, which displays data values.

Two surfaces, each contained in an hgtransform object to allow independent translation and overlap, construct the cursor. A third hgtransform object contains these two hgtransform objects as well as a text object. This third hgtransform (with handle `T` in the diagram and code) enables you to transform the cursor as a group.

The following diagram shows the containment hierarchy for this example. The axes contains a line, which is used to plot the data that the cursor moves along. The axes also contains the hierarchy of hgtransform objects that construct the cursor.



Note If you are using the MATLAB Help browser, run this example or open it in the MATLAB editor.

Set Up the Axes and Figure

The first step is to create an axes with fixed limits so MATLAB does not rescale the limits as the cursor moves along the line. Creating the axes automatically creates a figure to contain it.

Set figure properties to use the OpenGL renderer:

```
h_axes = axes('XLim',[-10 10],'YLim',[-5 5]);
set(get(h_axes,'Parent'),'Renderer','opengl')
```

Define the Transform Matrices and hgtransform Objects

The cross part of the cursor is formed from two surface objects, which are translated to overlap. Each surface is contained in its own hgtransform object (handles `t1` and `t2`) because they are translated in different directions. Both hgtransform objects are themselves contained in another hgtransform object (handle `T`).

See `makehgtform`, `hgtransform`.

```
% Create transform matrices
```

```
tmtx1 = makehgtform('translate',[-.5 0 0]);
tmtx2 = makehgtform('translate',[0 -.5 0]);

% Create hgtransform objects
T = hgtransform; % Contains the cursor
t1 = hgtransform('Parent',T,'Matrix',tmtx1);
t2 = hgtransform('Parent',T,'Matrix',tmtx2);
```

Create the Surface and Text Objects

The cursor is composed of two surface objects and a text object (to display data values). The two surfaces are parented to their respective hgtransform objects. The text is parented directly to the top-level hgtransform. The text object does not need coordinates because it is translated along with the surfaces in the top-level hgtransform object (T).

See cylinder, surface, text.

```
% Define surfaces and text
[sx,sy,sz] = cylinder([0 2 0]); % Use cylinder to generate data
surface(sz,sy,sx,'FaceColor','green',...
                'EdgeColor','none','FaceAlpha',.2,'Parent',t1);
surface(sx,sz./1.5,sy,'FaceColor','blue',...
                'EdgeColor','none','FaceAlpha',.2,'Parent',t2);
h_text = text('FontSize',12,'FontWeight','bold',...
                'HorizontalAlignment','center',...
                'VerticalAlignment','Cap','Parent',T);
```

Generate Data and Plot a Line

This example uses a line plot of a mathematical function to create a path along which to move the cursor.

```
% Plot the data x, y, and z
x = -10:.05:10;
y = [cos(x) + exp(-.01*x).*cos(x) + exp(.07*x).*sin(3*x)];
z = 1:length(x);
line(x,y,z)
```

Translate the Cursor Along the Plotted Line

To move the cursor along the line, a new transform matrix is calculated using each set of x, y, and z data points and used to set the Matrix property of the top-level hgtransfrom T. At the same time, the text object String property is updated to display the value of the current y data point.

The surfaces and the text translate together because they are allcontained in the top-level hgtransform object.

```
% Loop through the line data to move the cursor
for ind = 1:length(x)
    set(T,'Matrix',...
   makehgtform('translate',[x(ind) y(ind) z(ind)]))
    set(h_text,'String',num2str(y(ind)))
    drawnow,pause(.01)
end
```

Controlling Legends

In this section…

Legend Control Options

Graphics objects that represent data, such as lines, surfaces, patches, etc., can be represented in figure legends (see legend for information on creating legends). By setting object properties, you can:

- Include a particular graphics object in the legend (the default).
- Exclude a particular graphics object from the legend.
- Group graphics object together by parenting them to an hggroup or hgtransform object and represent the group as a single item in the legend ("Group Objects" on page 8-28).
- Display only the children of an object and not the parent in the legend. This is useful when the graph contains plot objects ("Plot Objects" on page 8-19).
- Specify the text label used in the legend for any object.

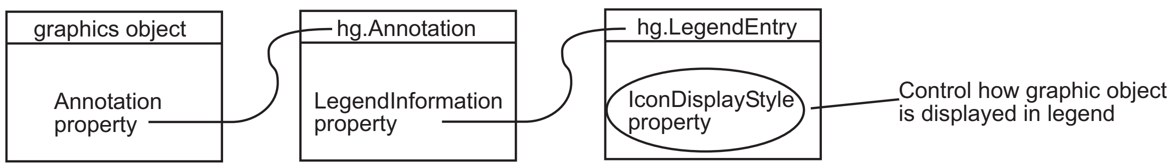
Properties for Controlling Legend Content

Graphics objects have two properties that control these options:

- `Annotation` — Controls whether the graphics object appears in the legend and determines if the object or its children appear in the legend.

- `DisplayName` — Specifies the text label used by the legend for the object. However, specifying a string with the legend commands resets the value of `DisplayName` property.

Accessing the Annotation Control Objects



Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object. The `hg.LegendEntry` object has a property called `IconDisplayStyle` that you can set to one of three values.

IconDisplayStyle Value	Behavior
`on`	Represent this object in a figure legend.
`off`	Do not include this object in a figure legend .
`children`	Display legend entries for this object's children and not the object itself (applies only to objects that have children, otherwise, the same as `on`).

For example, if `object_handle` is the handle of a graphics object, use the following statements to set the object's `IconDisplayStyle`. In this case, the graphics object, `object_handle`, is not included in the legend because its `IconDisplayStyle` property is set to `off`.

```
hAnnotation = get(object_handle,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Updating a Legend

If a legend exist and you change its `IconDisplayStyle` setting, you must call `legend` to update the display. See the `legend` command for the options available.

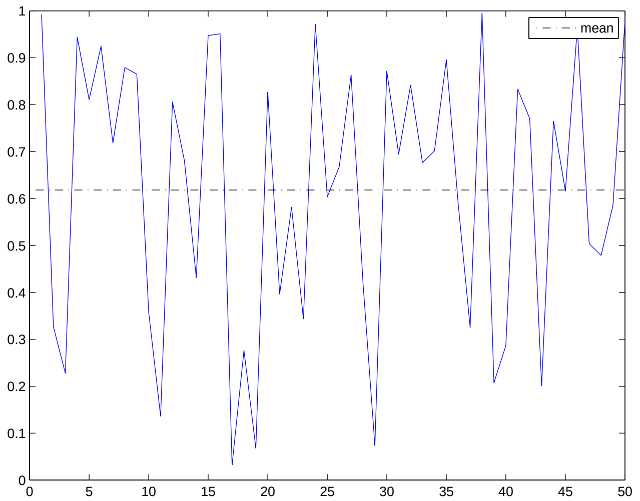
Example — Excluding a Particular Object From a Legend

This example creates a graph of random data values and draws a line at the mean *y* value. The blue data line does not appear in the legend because that line object has its `IconDisplayStyle` property of the associated `LegendEntry` object set to `off`. See "Properties for Controlling Legend Content" on page 8-100 for more information.

To execute the following code, copy it into a separate function file, save it as `annotation_property_line.m`, and run it from the Command Window.

```
function annotation_property_line
dat = rand(50,1);
hLine = plot(dat);
plotMean % Nested function draws a line at mean value
set(get(get(hLine,'Annotation'),'LegendInformation'),...
    'IconDisplayStyle','off'); % Exclude line from legend
legend('mean')
    function plotMean
    xlimits = get(gca,'XLim');
    meanValue = mean(dat);
    meanLine = line([xlimits(1) xlimits(2)],...
   [meanValue meanValue],'Color','k','LineStyle','-.');
    end
end
```

Here is the resulting graph.



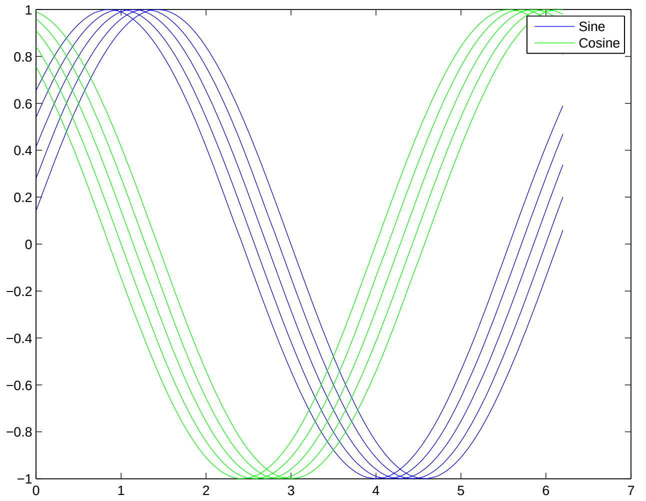
Example — One Legend Entry for a Group of Objects

You can group graphics objects in an hggroup or hgtransform object and represent the whole group as one item in a legend. This example creates two series of graphs (sines and cosines of the same data).

- The lines drawn to represent the sine are parented to one hggroup object.
- The lines drawn to represent the cosine are parented to another hggroup object.
- Both hggroup objects need their associated `IconDisplayStyle` property set to `on`.
- The legend then displays entries for both hggroup objects, but not their children (the plotted lines).

```
t = 0:.1:2*pi;
for k=1:5
    offset = k/7;
    m(:,k) = t+offset';
end
```

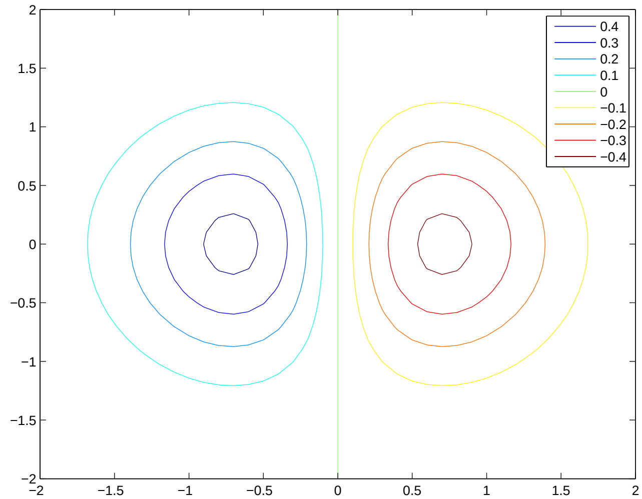
```
hSLines = plot(t,sin(m),'Color','b');hold on
hCLines = plot(t,cos(m),'Color','g');
hSGroup = hggroup;
hCGroup = hggroup;
set(hSLines,'Parent',hSGroup)
set(hCLines,'Parent',hCGroup)
% Include these hggroups in the legend:
set(get(get(hSGroup,'Annotation'),'LegendInformation'),...
    'IconDisplayStyle','on');
set(get(get(hCGroup,'Annotation'),'LegendInformation'),...
    'IconDisplayStyle','on');
legend('Sine','Cosine')
```



Example — Showing Children of Group Objects in Legend

You can include the children of a group in the legend by setting the group object's `IconDisplayStyle` to `children`. This is useful when graphs contain plot objects, which are groups of core graphics objects. For example, consider the following contour graph:

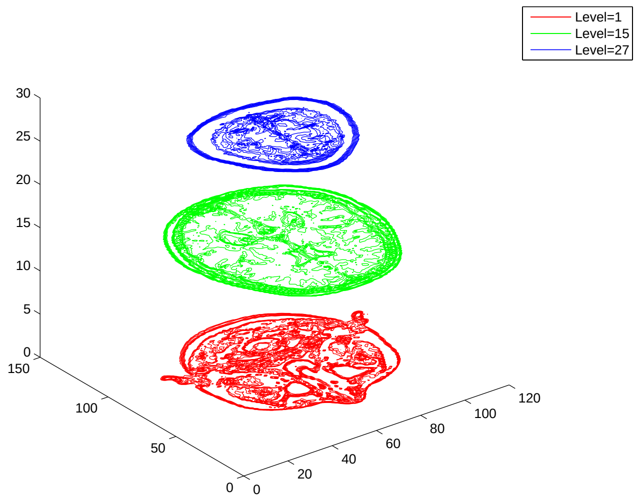
```
[X,Y] = meshgrid(-2:.1:2);
Z = X.*exp(-X.^2-Y.^2);
[mC hC] = contour(X,Y,Z);
set(get(get(hC,'Annotation'),'LegendInformation'),...
    'IconDisplayStyle','Children');
%{
Assigns each line object's DisplayName property a string
based on the value of the contour interval it represents
%}
k =1; ind = 1; hLines = get(hC,'Children');
while k < size(mC,2),
   set(hLines(ind),'DisplayName',num2str(mC(1,k)))
   k = k+mC(2,k)+1; ind = ind+1;
end
% Display the legend using DisplayName labels
legend('show')
```



Example — Grouping Objects to Reduce the Legend Entries

Some functions that visualize large data sets create many objects to render graphs. For example, `contourslice` uses patch objects to generate contour slices of volume data. This example groups the 1829 patch objects into hggroup objects according to which plane the objects represent and sets corresponding values for the `DisplayName` property, resulting in a legend with only three items.

```
load mri
D = squeeze(D);
phandles = contourslice(D,[],[],[1,15,27],8);view(3)
gh(1) = hggroup; gh(2) = hggroup; gh(3) = hggroup;
%set(gh,'Parent',gca)
for k=1:length(phandles)
   zd = get(phandles(k),'ZData');
   plane = num2str(zd(1));
   switch plane
      case '1'
         set(phandles(k),'Parent',gh(1),'EdgeColor','r')
      case '15'
         set(phandles(k),'Parent',gh(2),'EdgeColor','g')
      case '27'
         set(phandles(k),'Parent',gh(3),'EdgeColor','b')
      otherwise
         disp('Don''t know what to do with it')
   end
end
hA = get(gh,'Annotation');
hLL = get([hA{:}],'LegendInformation');
set([hLL{:}],{'IconDisplayStyle'},...
   {'on','on','on'}')
set(gh,{'DisplayName'},{'Level=1','Level=15','Level=27'}')
legend show
```


Level=1
Level=15
Level=27
30
25
20
15
10
5
0
150
100
50
0
0
20
40
60
80
100
120

Callback Properties for Graphics Objects

In this section...

What is a Callback?

A *callback* is a function that executes when a specific event occurs on a graphics object. You specify a callback by setting the appropriate property of the object. This section describes the events (specified via properties) for which you can define callbacks. See "Function Handle Callbacks" on page 8-110 for information on how to define callbacks.

Graphics Object Callbacks

All graphics objects have three properties for which you can define callback routines:

- `ButtonDownFcn` — Executes when users click the left mouse button while the cursor is over the object or within a 5-pixel border around the object.
- `CreateFcn` — Executes during object creation after all properties are set.
- `DeleteFcn` — Executes just before deleting the object.

User Interface Object Callbacks

User interface objects (e.g., uicontrol and uimenu) have a `Callback` property through which you define the function to execute when users activate these devices (e.g., click a push button or select a menu).

Figure Callbacks

Figures have additional properties that execute callback routines with the appropriate user action. Only the `CloseRequestFcn` property has a callback defined by default:

- `CloseRequestFcn` — Executes when a request is made to close the figure (by a `close` command, by the window manager menu, or by quitting MATLAB software).
- `KeyPressFcn` — Executes when users press a key while the cursor is within the figure window.
- `ResizeFcn` — Executes when users resize the figure window.
- `WindowButtonDownFcn` — Executes when users click a mouse button while the cursor is over the figure background, a disabled uicontrol, or the axes background.
- `WindowButtonMotionFcn` — Executes when users move the mouse button within the figure window (but not over menus or title bar).
- `WindowButtonUpFcn` — Executes when users release the mouse button, after having pressed the mouse button within the figure.

Function Handle Callbacks

In this section...

Introduction

Handle Graphics objects have a number of properties for which you can define callback functions. When a specific event occurs (e.g., a user clicks a push button or deletes a figure), the corresponding callback function executes. You can specify the value of a callback property as a

- String that is a MATLAB command or the name of an M-file
- Cell array of strings
- Function handle or a cell array containing a function handle and additional arguments

The following sections illustrate how to define function handle callbacks for Handle Graphics objects.

- "Introduction" on page 8-110 describes how to define a function handle callback.
- "Why Use Function Handle Callbacks" on page 8-112 provides information on the advantages of using function handle callbacks.
- "Example — Using Function Handles in GUIs" on page 8-114 shows how to create a simple GUI that uses function handle callbacks.

For general information on function handles, see the function handle reference page.

Function Handle Syntax

In Handle Graphics technology, functions that you want to use as function handle callbacks must define at least two input arguments in the function definition:

- The handle of the object generating the callback (the source of the event)
- The event data structure (can be empty for some callbacks)

MATLAB passes these two arguments implicitly whenever the callback executes. For example, consider the following statements, which are in a single M-file.

```
function myGui
% Create a figure and specify a callback
figure('WindowButtonDownFcn',@myCallback)
    .
    .
    .
% Callback subfunction defines two input arguments
function myCallback(src,eventdata)
    .
    .
    .
```

The first statement creates a figure and assigns a function handle to its `WindowButtondownFcn` property (created by using the `@` symbol before the function name). This function handle points to the subfunction `myCallback`. The definition of `myCallback` must specify the two required input arguments in its function definition line.

Passing Additional Input Arguments

You can define the callback function to accept additional input arguments by adding them to the function definition:

```
function myCallback(src,eventdata,arg1,arg2)
```

When using additional arguments for the callback function, you must set the value of the property to a cell array (i.e., enclose the function handle and arguments in curly braces):

```
figure('WindowButtonDownFcn',{@myCallback,arg1,arg2})
```

Defining Callbacks as a Cell Array of Strings — Special Case

Defining a callback as a cell array of strings is a special case because MATLAB treats it differently from a simple string. Setting a callback property to a string causes MATLAB to evaluate that string in the base workspace when the callback is invoked. However, setting a callback to a cell array of strings requires the following:

- The cell array must contain the name of an M-file that is on the MATLAB path as the first string element.
- The M-file callback must define at least two arguments (the handle of the callback object and an empty matrix).
- Any additional strings in the cell array are passed to the M-file callback as arguments.

For example,

```
figure('WindowButtonDownFcn',{myCallback,arg1})
```

requires you to define a function M-file that uses three arguments:

```
function myCallback(src,eventdata,arg1)
```

Why Use Function Handle Callbacks

Using function handles to specify callbacks provides some advantages over the use of strings, which must be either MATLAB commands or the name of an M-file that will be on the MATLAB path at run time.

Single File for All Code

Function handles enable you to use a single M-file for all callbacks. This is particularly useful when you are creating graphical user interfaces, because you can include both the layout commands and callbacks in one file.

For information on how to access subfunctions, see “Calling Subfunctions”.

Keeping Variables in Scope

When MATLAB evaluates function handles, the same variables are in scope as when the function handle was created. (In contrast, callbacks specified as strings are evaluated in the base workspace.) This simplifies the process of managing global data, such as object handles, in a GUI.

For example, suppose you create a GUI with a list box that displays workspace variables and a push button whose callback creates a plot using the variables selected in the list box. The push button callback needs the handle of the list box to query the names of the selected variables. Here's what to do:

1 Create the list box and save the handle:

```
h_listbox = uicontrol('Style','listbox',... etc.);
```

2 Pass the list box handle to the push button's callback, which is defined in the same M-file:

```
h_plot_button = uicontrol('Style','pushbutton',...
 'Callback',{@plot_button_callback,h_listbox},...,etc.);
```

The handle of the list box is now available in the plot button's callback without relying on global variables or using `findobj` to search for the handle. See "Example — Using Function Handles in GUIs" on page 8-114 for an example that uses this technique.

Callback Object Handle and Event Data

MATLAB passes additional information to the callback when it executes. This information includes the handle of the callback object (the source of the callback event) and event data that is specific to the particular callback property.

For example, the event data returned for the figure `KeyPressFcn` property is a structure that contains information about which keys were pressed.

Information about the event data associated with any given callback property is included with the property's documentation. Use the Handle Graphics Property Browser to access property documentation.

Function Handles Stay in Scope

A function handle can point to a function that is not in scope at the time of execution. For example, the function can be a subfunction in another M-file.

For a general discussion of function handles, see the "Function Handles" and "Anonymous Functions" in the MATLAB documentation.

Example — Using Function Handles in GUIs

This example creates a simple GUI that plots workspace variables. It is defined in a single M-file that contains both the layout commands and the callbacks. This example uses function handles to specify callback functions. Callbacks are implemented as nested functions to reduce the need to pass variables as arguments.

Complete Example Code

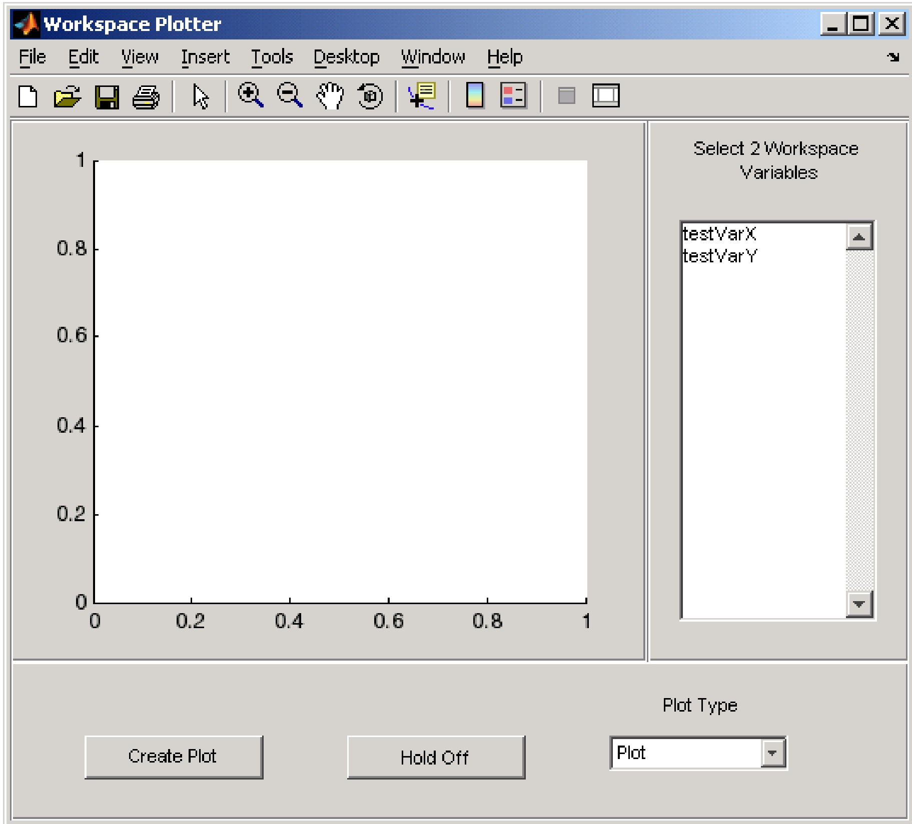
The documentation for this example does not list all the code used to lay out and program the GUI. To see a complete code listing, use the links in the Note.

Note If you are using the MATLAB Help browser, run this example or open it in the MATLAB editor.

See "Function Handle Callbacks" on page 8-110 for more information on the use of function handle callbacks.

The GUI Layout

The following graphic shows the GUI after running the example code. The program creates two variables (`testvarX` and `testVarY`) in the base workspace for testing purposes.



The GUI layout is split among three uipanel containers. The left-hand panel contains the axes, the right-hand panel contains a list box to display workspace variables, and the bottom panel contains the plot and hold buttons and the plot type pop-up menu.

Initialize the GUI

The list box and the hold toggle button need to be initialized before the GUI is ready to use. This is accomplished by executing their callbacks. Because you are calling these functions directly, MATLAB does not implicitly pass the first two arguments, as it would if these functions were executed as callbacks in response to an event. Therefore you must explicitly pass all arguments in these function calls:

```
% Initialize list box and make sure
% the hold toggle is set correctly
listBoxCallback(listBox,[])
holdToggleCallback(holdToggle,[])
```

The Callback Functions

The GUI components that have callbacks are the list box, toggle button, and plot push button. In addition, the figure's three uipanels define resize functions that MATLAB executes whenever users resize the figure.

See "Programming the Resize Functions" on page 8-93 for information on writing callback functions for the figure and uipanel `ResizeFcn` properties.

List Box Callback. The list box callback generates a list of the current variables in the base workspace using the `evalin` and `who` functions. It then assigns this list to the list box `String` property so that it displays these variable names.

The function takes advantage of the fact that the first argument passed to the callback is the handle of the callback object (i.e., the source of the callback event, which is the list box). Therefore, whenever you click in the list box, MATLAB updates the list to display the current workspace variables.

```
%% Callback for list box
function listBoxCallback(src,evt)
 % Load workspace vars into list box
 vars = evalin('base','who');
 set(src,'String',vars)
end % listBoxCallback
```

Plot Button Callback. The plot button callback performs three tasks:

- Gets the names of the variables selected by the user in the list box.
- Gets the type of plot selected by the user in the pop-up menu.
- Constructs and evaluates the plotting command in the base workspace:

  ```
  %% Callback for plot button
  function plotButtonCallback(src,evt)
   % Get workspace variables
  ```

```
    vars = get(listBox,'String');
    var_index = get(listBox,'Value');
    if length(var_index) ~= 2
     errordlg('You must select two variables',...
     'Incorrect Selection','modal')
     return
    end
    % Get data from base workspace
    x = evalin('base',vars{var_index(1)});
    y = evalin('base',vars{var_index(2)});
    % Get plotting command
    selected_cmd = get(popUp,'Value');
    % Make the GUI axes current and create plot
    axes(a)
    switch selected_cmd
    case 1 % user selected plot
     plot(x,y)
    case 2 % user selected bar
     bar(x,y)
    case 3 % user selected stem
     stem(x,y)
    end
   end % plotButtonCallback
```

Hold State Toggle Button Callback. The toggle button callback requires the handles of the GUI figure and axes. Because these callbacks are written as nested functions, the figure handle (`f`) and the axes handle (`a`) are in scope within the callback.

You want the GUI to toggle the hold state, but the GUI figure handle is hidden. It is necessary, therefore, to use the axes handle as the first argument to the `hold` function.

```
%% Callback for hold state toggle button
function holdToggleCallback(src,evt)
 button_state = get(src,'Value');
 if button_state == get(src,'Max')
  % toggle button is depressed
  hold(a,'on')
  set(src,'String','Hold On')
```

```
 elseif button_state == get(src,'Min')
  % toggle button is not depressed
  hold(a,'off')
  set(src,'String','Hold Off')
 end
end % holdToggleCallback
```

Optimizing Graphics Performance

In this section...

Introduction

This section discusses techniques that can help increase the speed with which MATLAB software creates graphs. These techniques apply to cases when you are creating many graphs of similar data and can improve rendering speed by preventing MATLAB from performing unnecessary operation.

Whether a given technique improves performance depends on the particular application. The `profile` function can help you determine where your code is consuming the most time.

General Performance Guidelines

The following list provides some general guidelines for optimizing performance:

- Set automatic-mode properties to manual whenever possible to prevent MATLAB from performing unnecessary operations.
- Modify existing objects instead of creating new ones.
- Use low-level core objects when creating objects repeatedly.
- Do not recreate legends or other annotations in a program loop; add these after you finish modifying the graph.

- Set the text `Interpreter` property to `none` if you are not using $\TeX$ characters.
- Try various renderers and erase modes. MATLAB might not have auto-selected the fastest renderer for your application.

The remainder of this section provides more details on these and other techniques.

- "Disabling Automatic Modes" on page 8-120 — Optimizing the use of axes objects.
- "Changing Graph Data Rapidly" on page 8-122 — Techniques for interactive plotting.
- "Performance of Bit-Mapped Images" on page 8-126 — Optimizing the use of image objects.
- "Performance of Patch Objects" on page 8-127 — Optimizing the use of patch objects.
- "Performance of Surface Objects" on page 8-128 — Optimizing the use of surface objects.

Disabling Automatic Modes

Graphics objects have properties that control many aspects of their behavior and appearance. The axes object in particular has many mode properties that enable it to respond to changes in the data represented in a graph.

For example, when you plot data, the axes determines appropriate axis limits, tick-mark placement, and labeling. Any changes you make to the plotted data (adding another line graph, for example) cause the axes to recompute the axis limits and to determine what values to use for the tick marks.

Fixing Axis Limits

The process of recalculating axis limits and the locations of the tick marks along each axis contributes to the time it takes to create a graph. If you are plotting data into the same axes repeatedly, you can improve performance by manually setting some or all of the axis limits, which, in turn, disables axis scaling and tick picking.

For example, suppose you are plotting time-series graphs in which you always view a 200-second time interval along the *x*-axis and your data ranges from –1 to 1. The following statement creates an axes with these limits and, in the process, sets the limit-picking mode to manual. Thereafter, MATLAB does not change the limits or recalculate tick mark locations (as long as you do not call a high-level plotting function like `plot`):

```
axes('XLim',[0 200],'YLim',[-1 1])
```

Setting All Modes to Manual

To maximize the efficiency with which MATLAB can update your graphs, disable all automatic operation so that MATLAB does not need to spend time determining if it is even necessary to recalculate a property value. The following steps illustrate this technique:

1. Create a figure and select the renderer you want to use. Line graphs should use `painters` to take advantage of its line thinning algorithm:

   ```
   figure('Renderer','painters')
   ```

   Setting a property automatically sets its associated mode property to manual.

2. Create an axes explicitly and set all properties (such as the axis limits) for which you can predetermine the appropriate value.

3. Set all other mode property values to `manual` (see the next table).

4. If you are creating line graphs using the `painters` renderer, set the axes `DrawMode` property to `fast`.

5. If you cannot determine the appropriate value for all mode properties, create your first graph and then use the `set` command to set mode properties to `manual`. See "Changing Graph Data Rapidly" on page 8-122 for an example.

The following table lists the axes mode properties and provides an explanation of what the mode controls.

Mode Property	What It Controls
ALimMode	Transparency limits mode
CameraPositionMode	Positioning of the viewpoint
CameraTargetMode	Positioning of the camera target in the axes
CameraUpVectorMode	The direction of “up” in 2-D and 3-D views
CameraViewAngleMode	The size of the projected scene and stretch-to-fit behavior
CLimMode	Mapping of data values to colors
DataAspectRatioMode	Relative scaling of data units along *x*-, *y*-, and *z*-axes and stretch-to-fit behavior
DrawMode	Controls the way MATLAB renders graphics objects (use with line graphs)
PlotBoxAspectRatioMode	Relative scaling of plot box along *x*-, *y*-, and *z*-axes and stretch-to-fit behavior
TickDirMode	Direction of axis tick marks (in for 2-D, out for 3-D)
XLimMode YLimMode ZLimMode	Limits of the respective *x*-, *y*-, and *z*-axes
XTickMode YTickMode ZTickMode	Tick mark spacing along the respective *x*-, *y*-, and *z*-axes
XTickLabelMode YTickLabelMode ZTickLabelMode	Tick mark labels along the respective *x*-, *y*-, and *z*-axes

Changing Graph Data Rapidly

MATLAB plotting functions perform a wide variety of operations in the process of creating a graph to make plotting easier. For example, the `plot`

function clears the current axes before drawing new lines, selects a line color or a marker type, searches for user-defined default values, and so on.

Low-Level Functions for Speed

The features that make plotting functions easy to use also consume computer resources. If you want to maximize graphing performance, use low-level functions and disable certain automatic features.

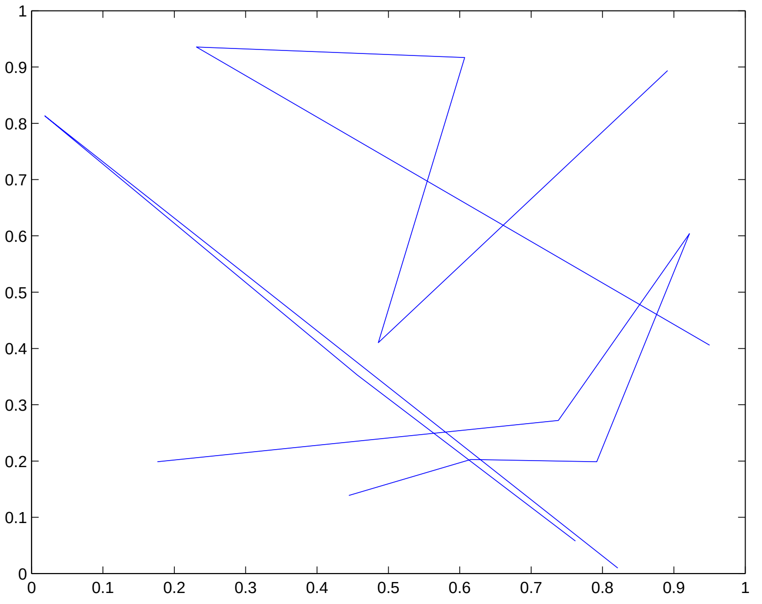
Low-level graphics functions (e.g., `line` vs. `plot`, `surface` vs. `surf`) perform fewer operation and therefore can be faster when you are creating many graphics objects. See "High-Level Versus Low-Level Functions" on page 8-17 for more information on how these functions differ.

Avoid Creating Graphics Objects

Each graphics object requires a certain amount of the computer's resources to create and store information, such as the value of all the object's properties. It is more efficient to use fewer graphics objects, if possible.

For example, add NaNs to vertex data (which causes that vertex to not be rendered) to create line segments that look like separate lines. Place the NaNs at identical locations in each vector of data:

```
x = [rand(5,1);nan;rand(4,1);nan;rand(6,1)];
y = [rand(5,1);nan;rand(4,1);nan;rand(6,1)];
line(x,y);
```



Update the Object's Data

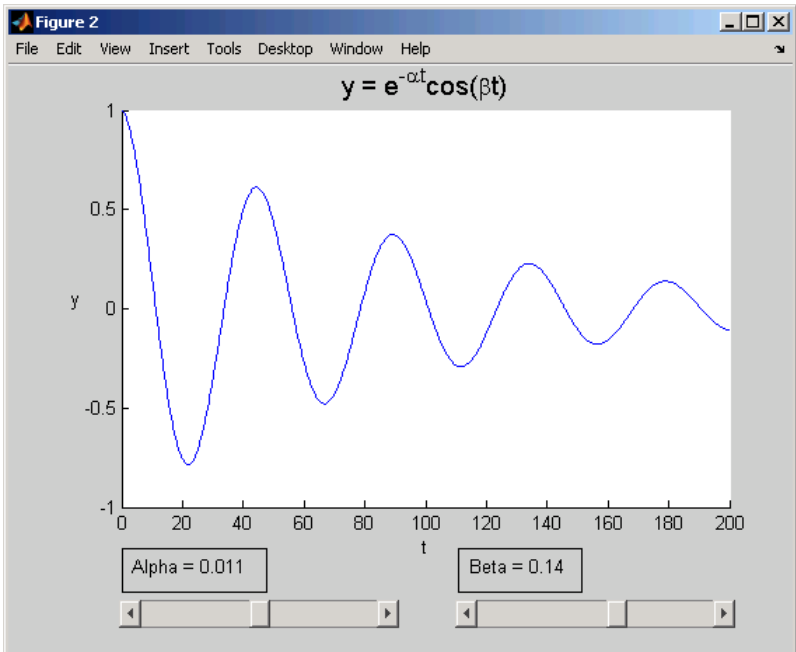
To view different data on what is basically the same graph, it is more efficient to update the data of the existing objects (lines, text, etc.) rather than recreating the entire graph.

For example, suppose you want to visualize the effect on your data of varying certain parameters. Follow these steps:

1. Set the limits of any axis that can be determined in advance, use `max` and `min` to determine the range of your data.
2. Recalculate the data using the new parameters.
3. Use the new data to update the data properties of the lines, text, etc. objects used in the graph.

4 Call `drawnow` to flush the event queue and update the figure (and all child objects in the figure).

The following example illustrates the use of these techniques in a GUI that uses sliders to vary two parameters in a mathematical expression, which is then plotted.



Note If you are using the MATLAB Help browser, run this example or open it in the MATLAB editor.

Specify Axes with Plotting Function for Better Performance

Most plotting functions accept an axes handle as an argument. This handle determines the target axes for the plot. Specifying the parent axes as an argument is often faster than using the `axes` function to make a particular

axes the current axes. For example, suppose you want six subplot in one figure and want to plot to each one in sequence:

```
x = 0:.05:2*pi;
for k=1:6
    h(k) = subplot(6,1,k);
end
for k=1:6
    y = sin(x*100*rand)*rand;
    plot(h(k),x,y) % Specify target axes in plotting function
    % Avoid using axes(h(k); plot(x,y) in a loop
end
```

Keeping Track of the Target Figure and Axes

You can explicitly set the figure `CurrentAxes` property to avoid specifying the axes handle with a number of functions that can operate on the current axes by default. For example, in the following code, both `stem` and `axis` operate on the current axes if you do not specify an axes as an argument:

```
fHandle = figure;
for k=1:6
    h(k) = subplot(6,1,k);
end
x = 0:.1:2*pi;
for k=1:6
    y = sin(x*100*rand)*rand;
    % Explicitly set current axes as a figure property
    set(fHandle,'CurrentAxes',h(k)); stem(x,y);
    axis([0 6.28 -1 1])
end
```

Both techniques give better performance than calling the `axes` function to control the current axes.

Performance of Bit-Mapped Images

Images can be defined with lower precision (than `double`) values to reduce the total amount of data required. MATLAB performs many operations on nondouble data types so you can use smaller image data without converting

the data to type `double`. See "Working with 8-Bit and 16-Bit Images" on page 6-10 for more information.

Direct Color Mapping

Where possible, use indexed images, because indexed images can apply direct mapping of pixel values to colormap values (`CDataMapping` set to `direct`). With direct mapping, MATLAB does not need to scale the data and then map it to values in the colormap.

See the `CDataMapping` image property for more information.

Use Truecolor for Smaller Images

The use of truecolor (red, green, and blue values) eliminates the need for color mapping. However, with very large images, the data can be quite large and slow performance.

Direct Mapping of Transparency Values

If you are using an alphamap of transparency values, prescale the alpha data so you can use the most efficient alpha data mapping (`AlphaDataMapping` set to `none`).

See the `AlphaDataMapping` image property for more information.

Performance of Patch Objects

Improve the speed with which MATLAB renders patch objects using the following techniques.

Define Patch Faces as Triangles

If you are using patch objects that have many vertices per patch face, modify your data so that each face has only three vertices, but still looks like your original object. This eliminates the tessellation step from the rendering process.

Use Data Thinning

It is sometimes possible (or even desirable) to reduce the number of vertices in a patch and still produce the desired results.

See the `reducepatch` and `reducevolume` functions for more information.

Direct Color Mapping

Where possible, use direct color mapping for coloring patches. (`CDataMapping` set to `direct`). With direct mapping, MATLAB does not need to scale the data and then map it to values in the colormap.

See the `CDataMapping` patch property for more information.

Use Truecolor for Smaller Patches

The use of truecolor (red, green, and blue values) eliminates the need for color mapping. However, with very large patches the data can be quite large and slow performance.

Direct Mapping of Transparency Values

If you are using an alphamap of transparency values, prescale the alpha data so you can use the most efficient alpha data mapping (`AlphaDataMapping` set to `none`).

See the `AlphaDataMapping` patch property for more information.

Performance of Surface Objects

Improve the speed with which MATLAB renders surface objects using the following techniques.

Direct Color Mapping

Where possible, use direct color mapping for coloring surfaces (`CDataMapping` set to `direct`). With direct mapping, MATLAB does not need to scale the data and then map it to values in the colormap.

See the `CDataMapping` surface property for more information.

Use Truecolor for Smaller Surfaces

The use of truecolor (red, green, and blue values) eliminates the need for color mapping. However, with very large surfaces, the data can be quite large and slow performance.

Mapping of Transparency Values

If you are using an alphamap of transparency values, prescale the alpha data so you can use the most efficient alpha data mapping (`AlphaDataMapping` set to `none`)

See the `AlphaDataMapping` surface property for more information.

Use Texture-Mapped Face Color

If you are using surface objects in an animation or want to be able to pan and rotate them quickly, you can achieve better rendering performance with large surfaces by setting `EdgeColor` to `none` and `FaceColor` to `texture`.

This technique is particularly useful if you want a high resolution surface without creating an objects whose data is large and therefore, very slow to transform. For example:

```
h1 = surf(peaks(1000));
shading interp
cd1 = get(h1,'CData');
surf(peaks(24),'FaceColor','Texture','EdgeColor','none',...
 'CData',cd1)
```

Figure Properties

Figure Objects

Figure graphics objects are the windows in which MATLAB software displays graphics. Figure properties enable you to control many aspects of these windows, such as their size and position on the screen, the coloring of graphics objects displayed within them, and the scaling of printed pictures.

This section discusses some of the features that are implemented through figure properties and provides examples of how to use these features.

See Figure Properties for a description of each individual property.

Related Information About Figures

For more information about figures, see the following links:

- “Graphics Windows — the Figure” on page 8-6
- “Preparing Figures and Axes for Graphics” on page 8-72
- “Protecting Figures and Axes” on page 8-78
- “The Figure Close Request Function” on page 8-81
- “Using Panel Containers in Figures — Uipanels” on page 8-90
- “Programming the Resize Functions” on page 8-93
- “Figure Callbacks” on page 8-108
- “Introduction” on page 10-36
- “Displaying Multiple Plots per Figure” on page 4-2

Docking Figures in the Desktop

In this section...

Introduction

You can dock figures in the MATLAB desktop by clicking the **dock** button, , which appears on the right end of the menu bar. Once docked, figures are placed in a figure group container, which you can also dock and undock.

You can select from a variety of arrangements of the figures in the container. The following picture shows how to select various figure arrangements. Once docked, the figure container displays the toolbar and menubar of the figure with focus.

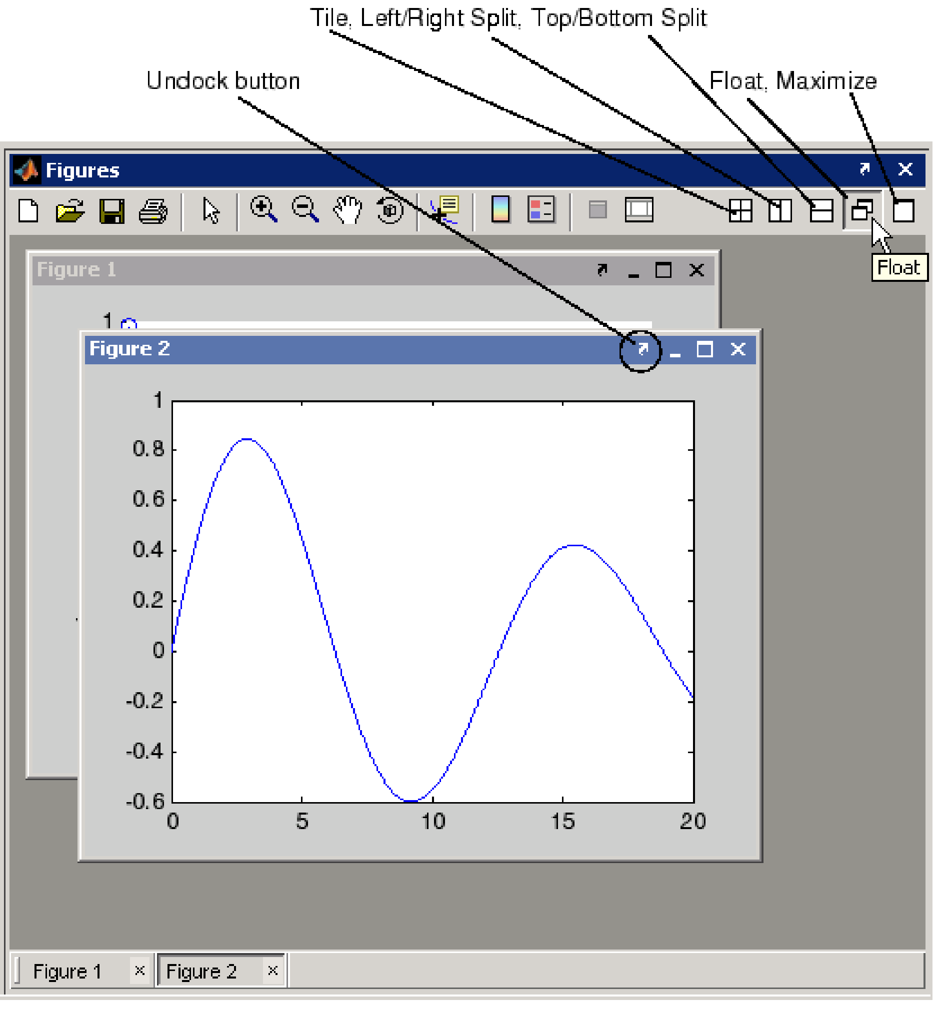


Figure Properties That Affect Docking

There are two figure properties that are related to figure docking — `DockControls` and `WindowStyle`.

DockControls

The `DockControls` property controls the display of the controls used to dock figures. Setting `DockControls` to `off` removes the **dock** button from the menubar and disables docking from the figure **Desktop** menu.

WindowStyle

When you set the `WindowStyle property` to `docked`, MATLAB docks the figure in the desktop within a figure group container.

If `WindowStyle` is set to `docked`,

- MATLAB automatically sets `DockControls` to `on`.
- You cannot set the `DockControls` property to `off`.
- You cannot set the figure `Position` property.

Docking Figures Automatically

If you want MATLAB to always dock figures, set the default value of the `WindowStyle` property to `docked`. The following statement,

```
set(0,'DefaultFigureWindowStyle','docked')
```

creates a default value for the `WindowStyle` property on the root level. Issuing this statement on the command line sets the `WindowStyle` of all figures for the duration of your MATLAB session (unless you change the value).

Place this statement in your `startup.m` file to make MATLAB always dock figures. See `startup` for more information on `startup.m`.

Creating a Nondockable Figure

In cases where you do not want users to be able to dock figures (e.g., figures used for GUIs), you should set figure properties as follows:

- `DockControls` to `off`
- `WindowStyle` to `normal` or `modal`
- `HandleVisibility` to `off` or `callback`

Positioning Figures

In this section...

Introduction

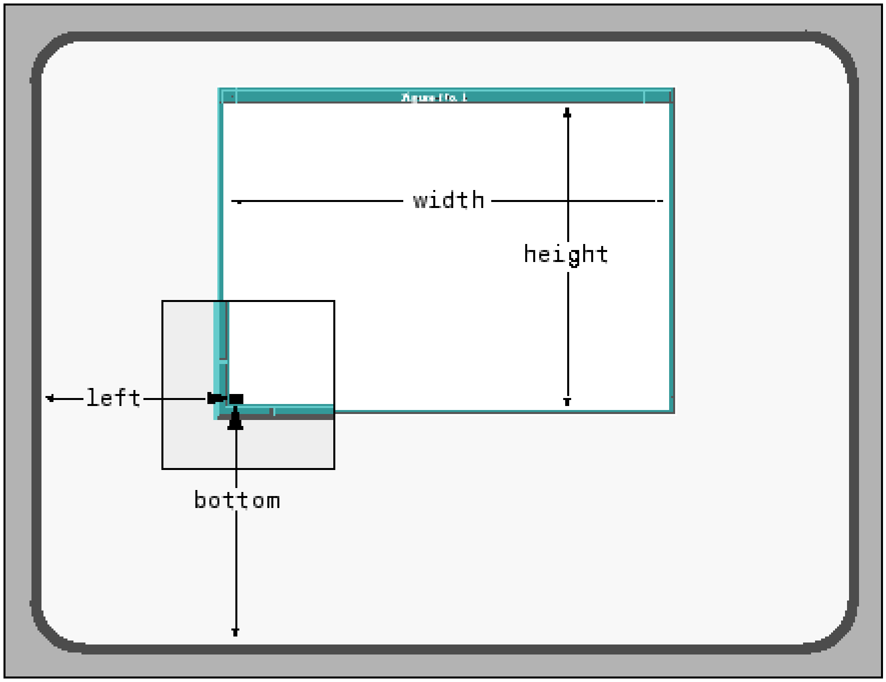
The figure `Position` property controls the size and location of the figure window on the screen. Monitor screen size is a property of the *root* Handle Graphics object. At startup, the MATLAB software determines the size of your computer screen and defines a default value for `Position`. This default creates figures about one-quarter of the screen's minimum extent and places them centered left to right, in the top half of the screen.

The Position Vector

MATLAB defines the figure `Position` property as a vector.

```
[left bottom width height]
```

`left` and `bottom` define the position of the first addressable pixel in the lower left corner of the window, specified with respect to the lower left corner of the screen. `width` and `height` define the size of the interior of the window (i.e., exclusive of the window border).



MATLAB does not measure the window border when placing the figure; the `Position` property defines only the internal active area of the figure window.

Because figures are windows under the control of your computer's windowing system, you can move and resize figures as you would any other windows. MATLAB automatically updates the `Position` property to the new values.

Figure Position and Window Managers

Your computer's window manager controls the layout of windows on monitors and on virtual desktops. It might not honor a request to place a figure window that would cause the entire figure or its top border to be located off-screen. A window manager also might force windows to have a certain minimum or maximum width or height. Such actions can cause a figure's position to differ from what you specify, and results can vary across platforms and window managers.

Figure Position for Docked Figures

When a figure is docked in the MATLAB desktop, the `Position` property is defined with respect to the figure group container within the desktop. See "Docking Figures in the Desktop" on page 9-3 for more information.

Units

The figure's `Units` property determines the units of the values with which you specify its position on the screen. Possible values for the `Units` property are

```
set(gcf,'Units')
[ inches | centimeters | normalized | points | {pixels} |
characters]
```

with `pixels` being the default. These choices allow you to specify the figure size and location in absolute units (such as inches) if you want the window always to be a certain size, or in units relative to the screen size (such as pixels).

Characters are units that enable you to define the location and size of the figure in units that are based on the size of the default system font. Ffor an example that uses character units, see "Example — Using Figure Panels" on page 8-91.

Determining Screen Size

Whatever units you use, it is important to know the extent of the screen in those units. You can obtain this information from the root `ScreenSize` property. For example:

```
get(0,'ScreenSize')
ans =
     1 1 1152 900
```

In this case, the screen is 1152 by 900 pixels. MATLAB returns the `ScreenSize` in the units determined by the root `Units` property. For example,

```
set(0,'Units','normalized')
```

normalizes the values returned by `ScreenSize`.

```
get(0,'ScreenSize')
ans =
    0 0 1 1
```

Defining the figure `Position` in terms of the `ScreenSize` in normalized units makes the specification independent of variations in screen size. This is useful if you are writing an M-file to use on different computer systems. It does, however, result in differently-shaped figures on monitors having different aspect ratios.

Example — Specifying Figure Position

Suppose you want to define two figure windows that occupy the upper third of the computer screen (e.g., one for uicontrols and the other for displaying data). To position the windows precisely, you must take into account the window borders (which can include a menu bar and toolbars) when calculating the size and offsets. For this purpose, use the windows' `OuterPosition` rather than their `Position` property.

1 Ensure root units are pixels and get the size of the screen and create a figure window:

```
set(0,'Units','pixels')
scnsize = get(0,'ScreenSize');
fig1 = figure;
```

2 The figure `Position` property only includes the drawable extent of the window, exclusive of the window borders. Obtain the entire window's size from the `OuterPosition` property, and compare the two:

```
position = get(fig1,'Position')
outerpos = get(fig1,'OuterPosition')
borders = outerpos - position

position =
   560   528   560   420

outerpos =
   556   524   568   495

borders =
```

```
-4    -4     8    75
```

The left, right, and bottom borders are each 4 pixels wide. The top border, which contains a menu bar and a figure toolbar is 75-4, or 71 pixels wide.

3 Create a second figure, which defaults to the same size as the first one:

```
fig2 = figure;
```

4 Define the desired size and location of the figures. Leave a space equal to their border width between them:

```
edge = -borders(1)/2;
pos1 = [edge,...
        scnsize(4) * (2/3),...
        scnsize(3)/2 - edge,...
        scnsize(4)/3];
pos2 = [scnsize(3)/2 + edge,...
        pos1(2),...
        pos1(3),...
        pos1(4)];
```

5 Reposition the two figures by changing both of their `OuterPosition` properties:

```
set(fig1,'OuterPosition',pos1)
set(fig2,'OuterPosition',pos2)
```

The two figures now occupy the top third of the screen.

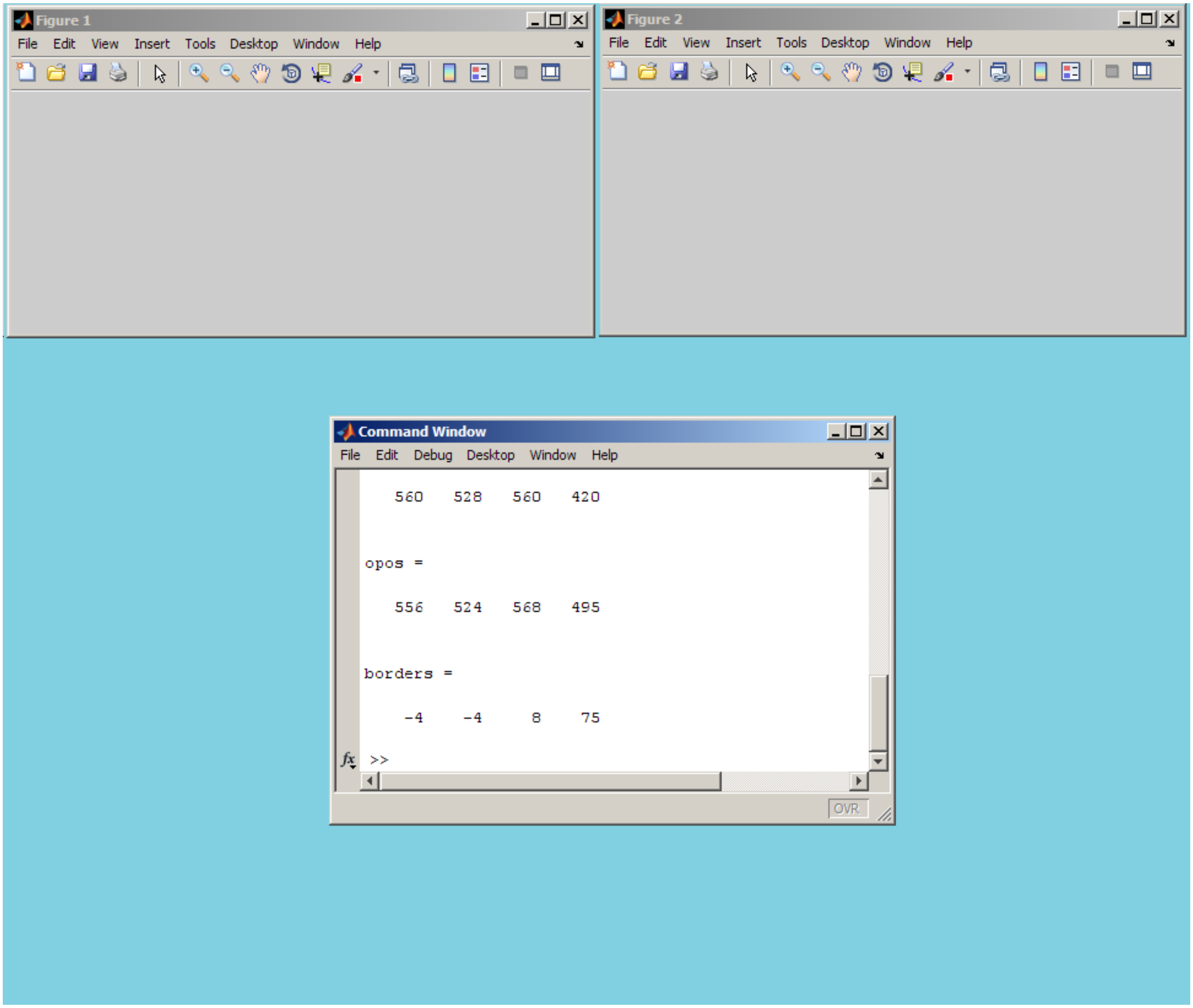

Figure 1
File Edit View Insert Tools Desktop Window Help
Figure 2
File Edit View Insert Tools Desktop Window Help
Command Window
File Edit Debug Desktop Window Help
560 528 560 420
opos =
556 524 568 495
borders =
-4 -4 8 75
>>
OVR

Figure Colormaps — The Colormap Property

In this section...

Introduction

The MATLAB software defines a colormap as a three-column array. Each row of the array defines a particular color by giving three values in the range [0...1]. These values specify the RGB values; the intensity of the red, green, and blue video components.

Colormaps enable you to control how MATLAB maps data values to colors in surfaces, patches, images, and plotting functions that are based on these objects. See the following sections for more information.

- "Coloring Mesh and Surface Plots" in 3-D Visualization
- "Specifying Patch Coloring" in 3-D Visualization
- "The Image Object and Its Properties" on page 6-27

Specifying the Figure Colormap

The figure `Colormap` property contains the colormap array. You can specify the figure colormap by setting this property to an *m*-by-3 array, where *m* is the number of colors in the colormap.

For example, the following statement creates a figure with a colormap having 128 random colors.

```
figure('Colormap',rand(128,3));
```

The `colormap` function is an easy way to specify the colormap. MATLAB also provides a number of functions that generate colormaps. For example,

```
colormap(hsv(96))
```

sets the colormap of the current figure to a 96 element version of the `hsv` colormap. See the `colormap` reference page for a list of predefined colormaps. The default colormap is `jet(64)`.

Selecting Drawing Methods

In this section...

Double Buffering

Overview

Set `DoubleBuffer` to `on` when you are animating lines rendered in `painters` with `EraseMode` set to `normal`.

More Details

Double buffering is the process of drawing into an offscreen pixel buffer and then blitting the buffer contents to the screen once the drawing is complete (instead of drawing directly to the screen, where the process of drawing is visible as it progresses). Double buffering generally produces flash-free rendering for simple animations (such as those involving lines, as opposed to objects containing large numbers of polygons).

The figure `DoubleBuffer` property accepts the values `on` and `off`, with `on` being the default. You can select double buffering only when the figure `Renderer` property is set to `painters`. `zbuffer` and `opengl` always use double buffering and ignore this property.

Use double buffering with the animated object's `EraseMode` property set to `normal`.

Selecting a Renderer

Overview

The MATLAB software automatically selects the best renderer based on the complexity of the graphics objects and the options available on your system.

More Details

A renderer is the software that processes graphics data (such as vertex coordinates) into a form that MATLAB can use to draw into the figure. MATLAB supports three renderers:

- Painters
- Z-buffer
- OpenGL

Painters

Painters method is faster when the figure contains only simple or small graphics. It cannot be used with lighting.

Z-Buffer

Z-buffering is the process of determining how to render each pixel by drawing only the front-most object, as opposed to drawing all objects back to front, redrawing objects that obscure those behind. The pixel data is buffered and then blitted to the screen all at once.

Z-buffering is generally faster for more complex graphics, but can be slower for very simple graphics. You can set the `Renderer` property to whatever produces the fastest drawing (either `zbuffer` or `painters`), or let MATLAB decide which method to use by setting the `RendererMode` property to `auto` (the default).

Printing from Z-Buffer. You can select the resolution of the PostScript file produced by the `print` command using the `-r` option. By default, MATLAB prints Z-buffered figures at a medium resolution of 150 dpi (the default with `Renderer` set to `painters` is 864 dpi).

The size of the file generated from a Z-buffer figure does not depend on its contents, just the size of the figure. To decrease the file size, make the `PaperPosition` property smaller before printing (or set `PaperPositionMode` to `auto` and resize the figure window).

OpenGL

OpenGL is available on many computer systems. It is generally faster than either painters or Z-buffer and in some cases enables MATLAB to use the system's graphics hardware (which results in significant speed increase). See the figure `Renderer` property for more information.

Limitations of OpenGL. OpenGL has two limitations when compared to painters and Z-buffer:

- OpenGL does not interpolate colors within the figure colormap; all color interpolation is performed through the RGB color cube, which can produce unexpected results.
- OpenGL does not support Phong lighting.

Specifying the Figure Pointer

In this section...

Predefined Figure Pointer Symbols

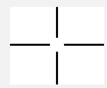
The MATLAB software indicates the position of the pointer (cursor) within the figure window using a graphical symbol. You can select a pointer from 15 predefined symbols (see table below) or you can define your own symbol. By convention, each of the predefined symbols has a purpose associated with it (although MATLAB enforces no rules for the use of any symbols).

You specify the pointer symbol by setting the value of the figure `Pointer` property. For example, this statement sets the pointer in the current figure (`gcf`) to an arrow.

```
set(gcf,'Pointer','arrow')
```

The following table shows the predefined symbols, the associated specifier, and describes typical use.

Purpose	Specifier	Typical Symbol
Use for editing text	`ibeam`	
Locate a point on a graphics object	`crosshair`	
Select a point anywhere in the figure	`arrow`	
Indicate the system is busy	`watch`	
Resize an object from the top-left corner	`topl`	
Resize an object from the top-right corner	`topr`	

Purpose	Specifier	Typical Symbol
Resize an object from the bottom-left corner	`botl`	
Resize an object from the bottom-right corner	`botr`	
View the actual hot spot	`circle`	
Locate a point	`cross`	
Use as popular symbol	`fleur`	
Resize an object from the left side	`left`	
Resize an object from the right side	`right`	
Resize an object from the top	`top`	
Resize an object from the bottom	`bottom`	
Align a point with other objects on the display	`fullcross`	
See the next section for information on defining your own pointer shape	`custom`	

Defining Custom Pointers

When you set the `Pointer` property to `custom`, MATLAB displays the pointer you define using the `PointerShapeCData` and the `PointerShapeHotSpot` properties. Custom pointers are 16-by-16 pixels, where each pixel can be either black, white, or transparent.

Specify the pointer by creating a 16-by-16 matrix containing elements that are

- 1's where you want the pixel black
- 2's where you want the pixel white
- `NaN`s where you want the pixel transparent

Assign the matrix to the figure PointerShapeCData property. MATLAB displays the defined pointer whenever the pointer is in the figure window.

The PointerShapeHotSpot property specifies the pixel that indicates the pointer location. MATLAB then stores this location in the root PointerLocation property. Set the PointerShapeHotSpot property to a two-element vector specifying the row and column indices in the PointerShapeCData matrix that correspond to the pixel specifying the location. The default value for this property is [1 1], which corresponds to the upper left corner of the pointer.

Example — Two Custom Pointers

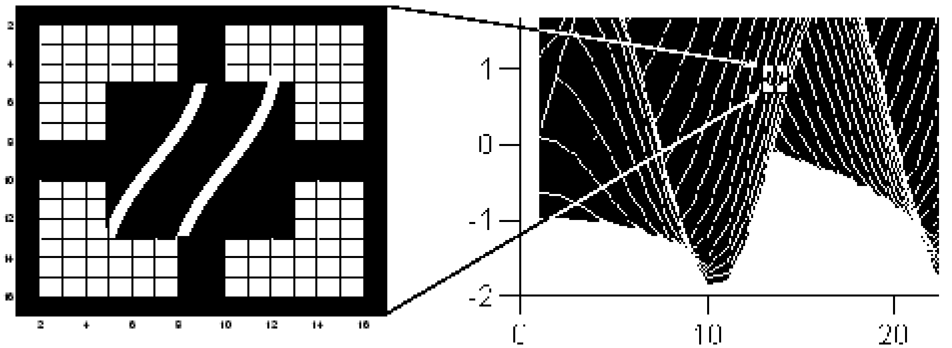
One way to create a custom pointer is to assign values to a 16-by-16 matrix by hand, as illustrated in the following example.

First, initialize the matrix, setting all values to 2. Create a black border 1 pixel wide. Add alignment marks.

```
P = ones(16)+1;
P(1,:) = 1; P(16,:) = 1;
P(:,1) = 1; P(:,16) = 1;
P(1:4,8:9) = 1; P(13:16,8:9) = 1;
P(8:9,1:4) = 1; P(8:9,13:16) = 1;
P(5:12,5:12) = NaN;  % Create a transparent region in the center
set(gcf,'Pointer','custom','PointerShapeCData',P,...
        'PointerShapeHotSpot',[9 9])
```

The last statement sets the Pointer property to custom, assigns the matrix to the PointerShapeCData property, and selects element (9,9) as the “hot spot.”

MATLAB now uses the custom pointer within the figure window.

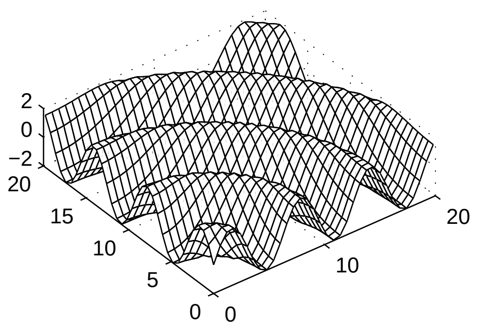


Creating Pointers from Functions. You can use a mathematical function to define the PointerShapeCData matrix. For example, evaluating the function

$$2(\sin(\sqrt{x^2 + y^2}))$$

```
g = 0:.2:20;
[X,Y] = meshgrid(g);
Z = 2*sin(sqrt(X.^2 + Y.^2));
mesh(Z);
```

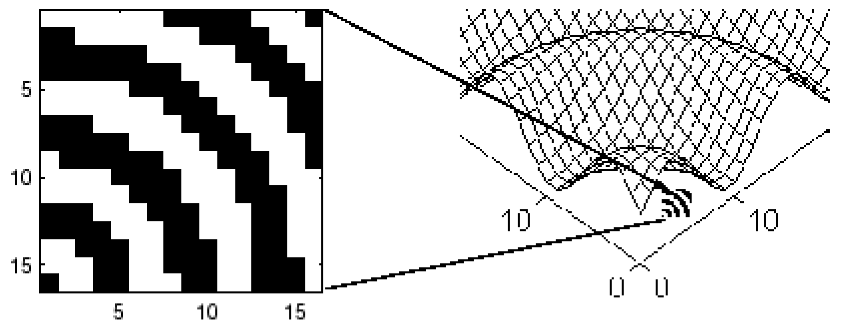
produces an interesting surface.



Use the values of Z to create a pointer sampling fewer points so that Z is a 16-by-16 matrix.

```
g = linspace(0,20,16);
[X,Y] = meshgrid(g);
Z = 2*sin(sqrt(X.^2 + Y.^2));
set(gcf,'Pointer','custom',...
        'PointerShapeCData',flipud((Z>0) + 1))
```

The statement `flipud((Z>0) + 1)` sets all values in `Z` that are greater than 0 to 2 (in MATLAB, true + 1 = 2), less than 0 to 1 (false + 1 = 1) and then flips the data around so that element (1,1) is the upper left corner.



Axes Properties

Axes Objects — Defining Coordinate Systems for Graphs

MATLAB software uses graphics objects to create visual representations of data. For example, a two-dimensional array of numbers can be represented as lines connecting the data points defined by each column, as a surface constructed from a grid of rectangles whose vertices are defined by each element of the array, as a contour graph where equal values in the array are connected by lines, and so on.

In all these cases, there must be a frame of reference that defines where to place each data point on the graph. This frame of reference is the coordinate system defined by the axes. Axes orient and scale graphs to produce the view of the data that you see on screen.

MATLAB creates axes to define the coordinate system of each graph. Axes are always contained within a figure object and themselves contain the graphics objects that make up the graph.

Axes properties control many aspects of how MATLAB displays graphical information. This section discusses some of the features that are implemented through axes properties and provides examples of how to uses these features.

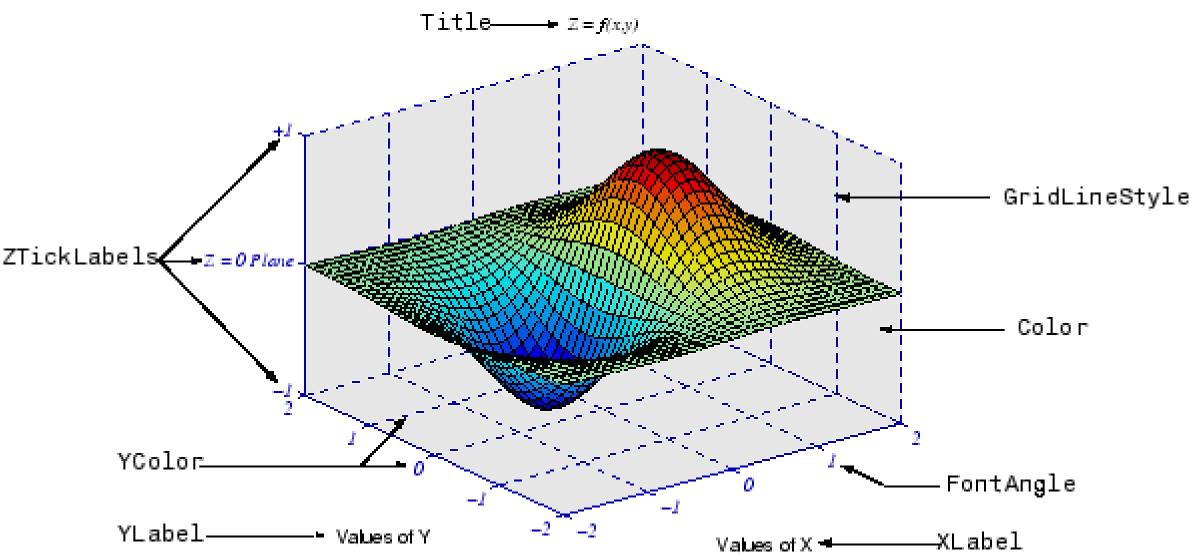
The Axes Properties list all axes properties and provide an overview of the characteristics that are affected by each property.

Labeling and Appearance Properties

In this section...

Introduction

MATLAB software provides a number of properties for labeling and controlling the appearance of axes. For example, this surface plot shows some of the labeling possibilities and indicates the controlling property.



Creating Axes with Specific Characteristics

To create this axes, specify values for the indicated properties.

```
h = axes('Color',[.9 .9 .9],...
        'GridLineStyle','--',...
        'ZTickLabel','-1|Z = 0 Plane|+1',...
```

```
'FontName','times',...
'FontAngle','italic',...
'FontSize',14,...
'XColor',[0 0 .7],...
'YColor',[0 0 .7],...
'ZColor',[0 0 .7]);
```

Axis Labels

The individual axis labels are text objects whose handles are normally hidden from the command line (their `HandleVisibility` property is set to `callback`). You can use the `xlabel`, `ylabel`, `zlabel`, and `title` functions to create axis labels. However, these functions affect only the current axes. If you are labeling axes other than the current axes by referencing the axes handle, then you must obtain the text object handle from the corresponding axes property.

Getting the Text Object Handle

For example,

```
get(axes_handle,'XLabel')
```

returns the handle of the text object used as the *x*-axis label. Obtaining the text handle from the axes is useful in M-files and MATLAB applications where you cannot be sure the intended target is the current axes.

The following statements define the *x*- and *y*-axis labels and title for the axes above.

```
set(get(axes_handle,'XLabel'),'String','Values of X')
set(get(axes_handle,'YLabel'),'String','Values of Y')
set(get(axes_handle,'Title'),'String','\fontname{times}\itZ =
f(x,y)')
```

Because the labels are text, you must specify a value for the `String` property, which is initially set to the empty string (i.e., there are no labels).

MATLAB overrides many of the other text properties to control positioning and orientation of these labels. However, you can set the `Color`, `FontAngle`, `FontName`, `FontSize`, `FontWeight`, and `String` properties.

Specifying Axis Label Fonts

Both axes objects and text objects have font specification properties. The call to the `axes` function on the previous page set values for the `FontName`, `FontAngle`, and `FontSize` properties.

If you want to specify the font for the labels and title, set the font property values when defining their `String` property. For example, the *x*-axis label statement would be

```
set(get(h,'XLabel'),'String','Values of X',...
                    'FontName','times',...
                    'FontAngle','italic',...
                    'FontSize',14)
```

Bitmapped Vs. Truetype Fonts — Text Does Not Rotate

Bitmapped fonts (e.g., Courier) cannot be rotated on the display. Therefore, when you specify a bitmapped font with the `FontName` property, this text might not be rotated correctly, for example, when used as the *y*-axis label.

To avoid problems with bitmapped fonts, use TrueType fonts. For example, you might have a TrueType font named Courier New that you can use instead of Courier. See your system documentation for information on which fonts are installed on your system.

Positioning Axes

In this section…

Introduction

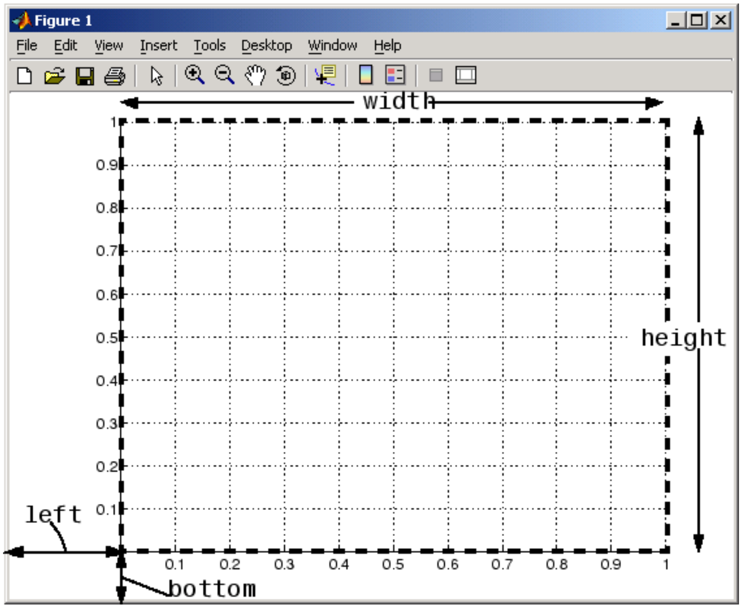
The axes `Position` property controls the size and location of an axes within a figure. The default axes has the same aspect ratio (ratio of width to height) as the default figure and fills most of the figure, leaving a border around the edges. However, you can define the axes position as any rectangle and place it wherever you want within a figure.

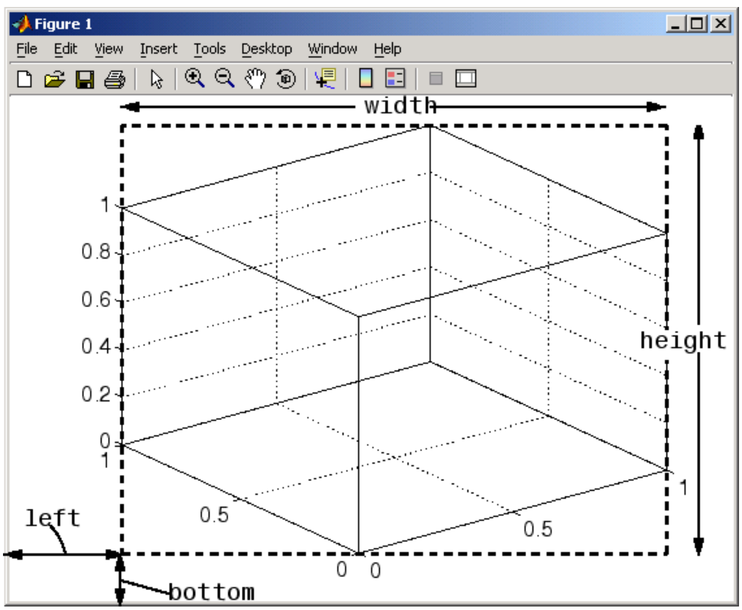
The Position Vector

The MATLAB software defines the axes `Position` property as a vector.

```
[left bottom width height]
```

`left` and `bottom` define a point in the figure that locates the lower left corner of the axes rectangle. `width` and `height` specify the dimensions of the axes rectangle. Viewing the axes in 2-D (azimuth = 0°, elevation = 90°) orients the *x*-axis horizontally and the *y*-axis vertically. From this angle, the plot box (the area used for plotting, exclusive of the axis labels) coincides with the axes rectangle.


Figure 1
File Edit View Insert Tools Desktop Window Help
width
height
left
bottom
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1


Figure 1
File Edit View Insert Tools Desktop Window Help
width
height
left
bottom
1
0.8
0.6
0.4
0.2
0
1
0.5
0 0
0.5
1

By default, MATLAB draws the plot box to fill the axes rectangle, regardless of its shape. However, axes properties enable control over the shape and scaling of the plot box.

Position Units

The axes `Units` property determines the units of measurement for the `Position` property. Possible values for this property are

```
set(gca,'Units')
[ inches | centimeters | {normalized} | points | pixels ]
```

with `normalized` being the default. Normalized units map the lower left corner of the figure to the point (0,0) and the upper right corner to (1.0,1.0), regardless of the size of the figure. Normalized units cause axes to resize automatically whenever you resize the figure. All other units are absolute measurements that remained fixed as you resize the figure.

Automatic Axes Resize

In this section...

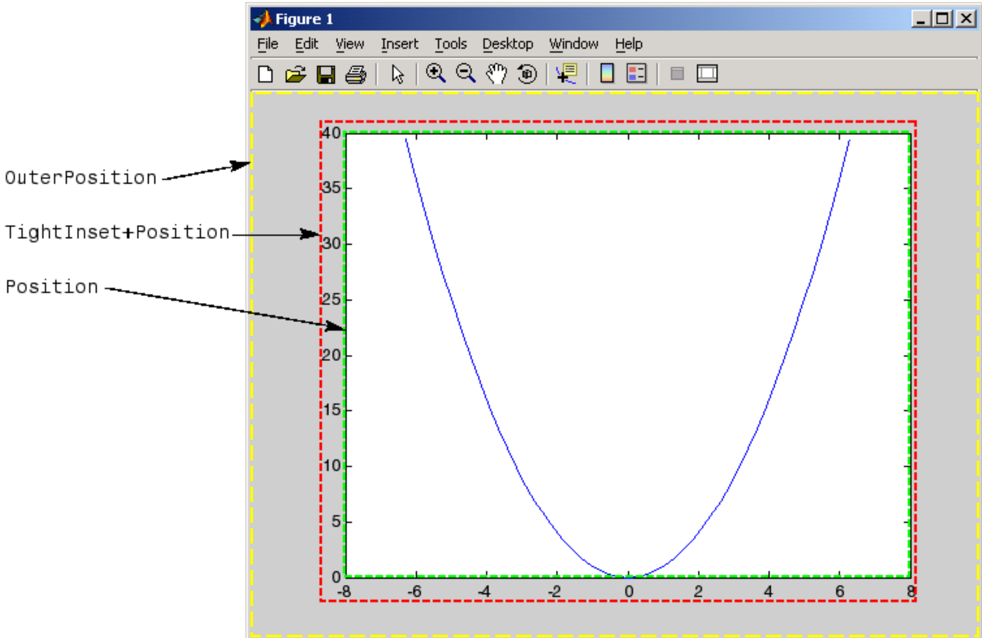
Properties Controlling Axes Size

When you create a graph, the MATLAB software automatically creates an axes to display the graph. The axes is sized to fit in the figure and automatically resizes as you resize the figure. However, MATLAB applies the automatic resize behavior only when the axes `Units` property is set to `normalized` (the default).

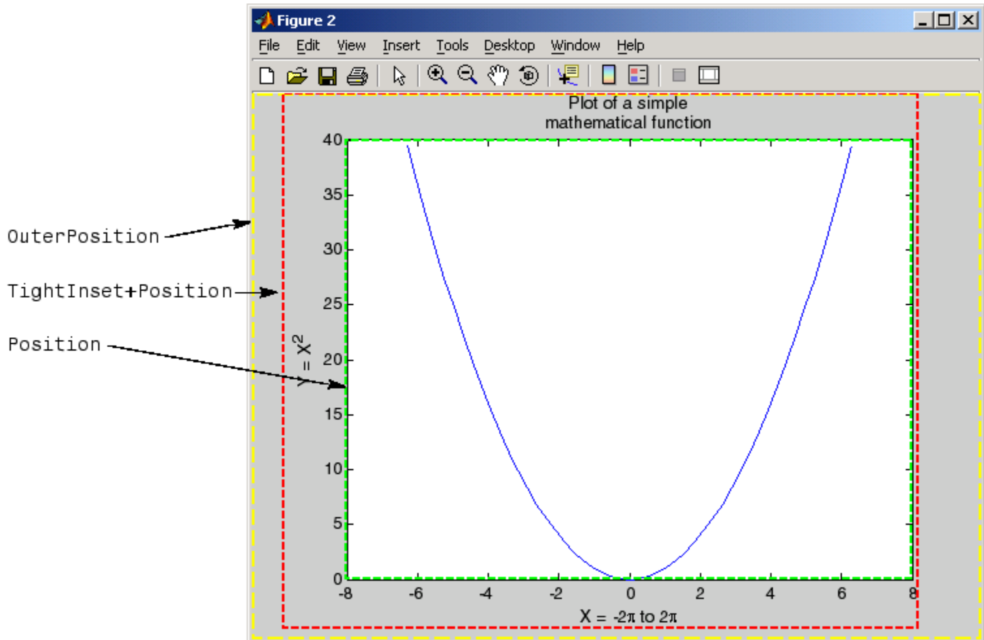
You can control the resize behavior of the axes using the following axes properties.

- `OuterPosition` — The boundary of the axes including the axis labels, title, and a margin. For figures with only one axes, this is the interior of the figure.
- `Position` — The boundary of the axes, excluding the tick marks and labels, title, and axis labels.
- `ActivePositionProperty` — Specifies whether to use the `OuterPosition` or the `Position` property as the size to preserve when resizing the figure containing the axes.
- `TightInset` — The margins added to the width and height of the `Position` property to include text labels, title, and axis labels.
- `Units` — Keep this property set to `normalized` to enable automatic axes resizing.

The following graph shows the areas defined by the `OuterPosition`, `TightInset + Position`, and `Position` properties.



When you add axis labels and a title, the `TightInset` changes to accommodate the additional text, as shown in the following graph.

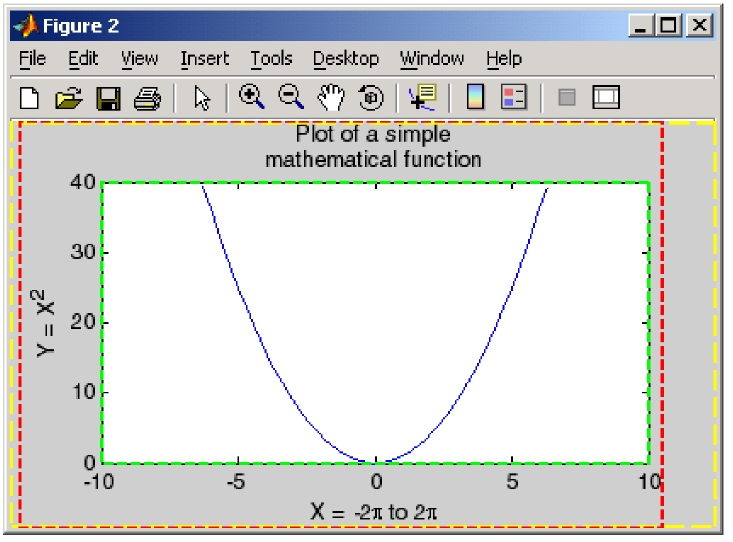


Now the size of the rectangle defined by the `TightInset` + `Position` properties includes all graph text. The `Position` and `OuterPosition` properties remain unchanged.

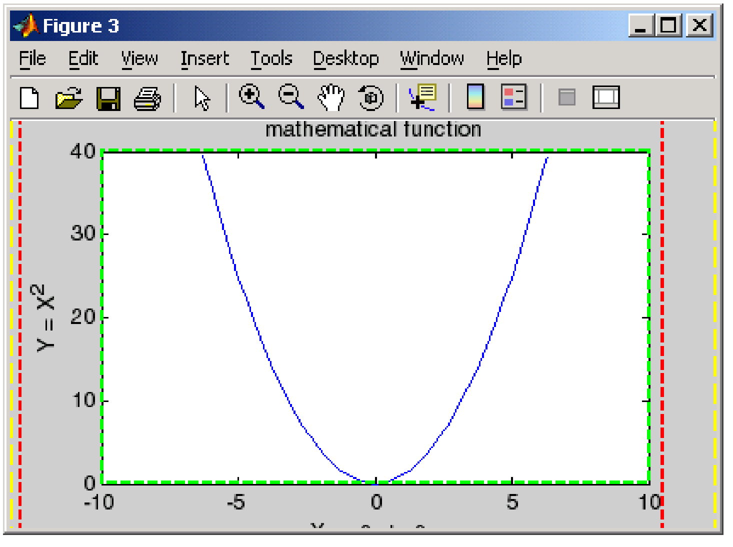
Using OuterPosition as the ActivePositionProperty

As you resize the figure, MATLAB maintains the area defined by the `TightInset` + `Position` so the text is not cut off. Compare the next two graphs, which have both been resized to the same figure size.

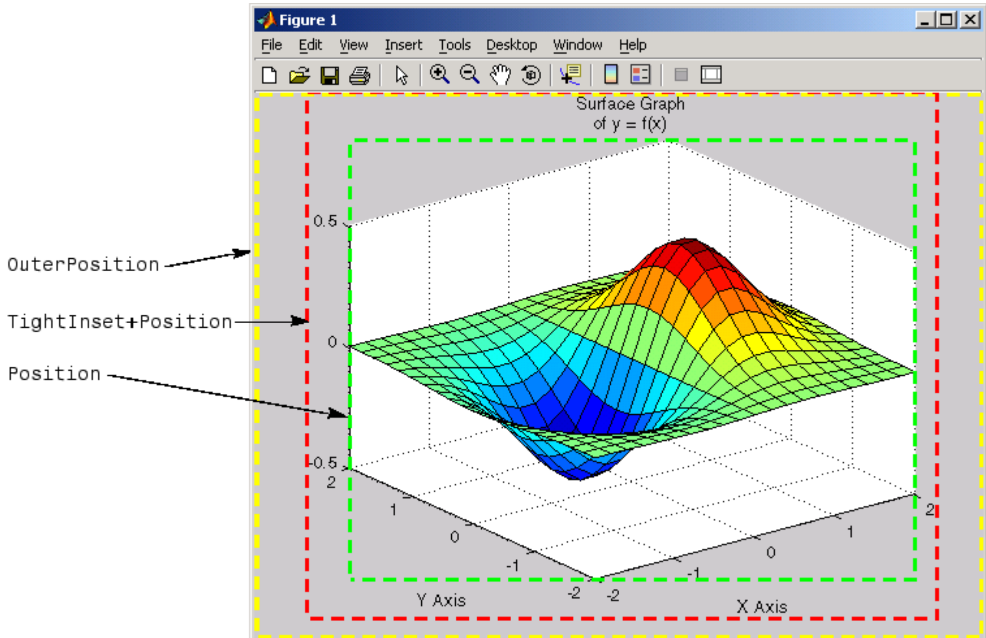
ActivePositionProperty = OuterPosition



ActivePositionProperty = Position



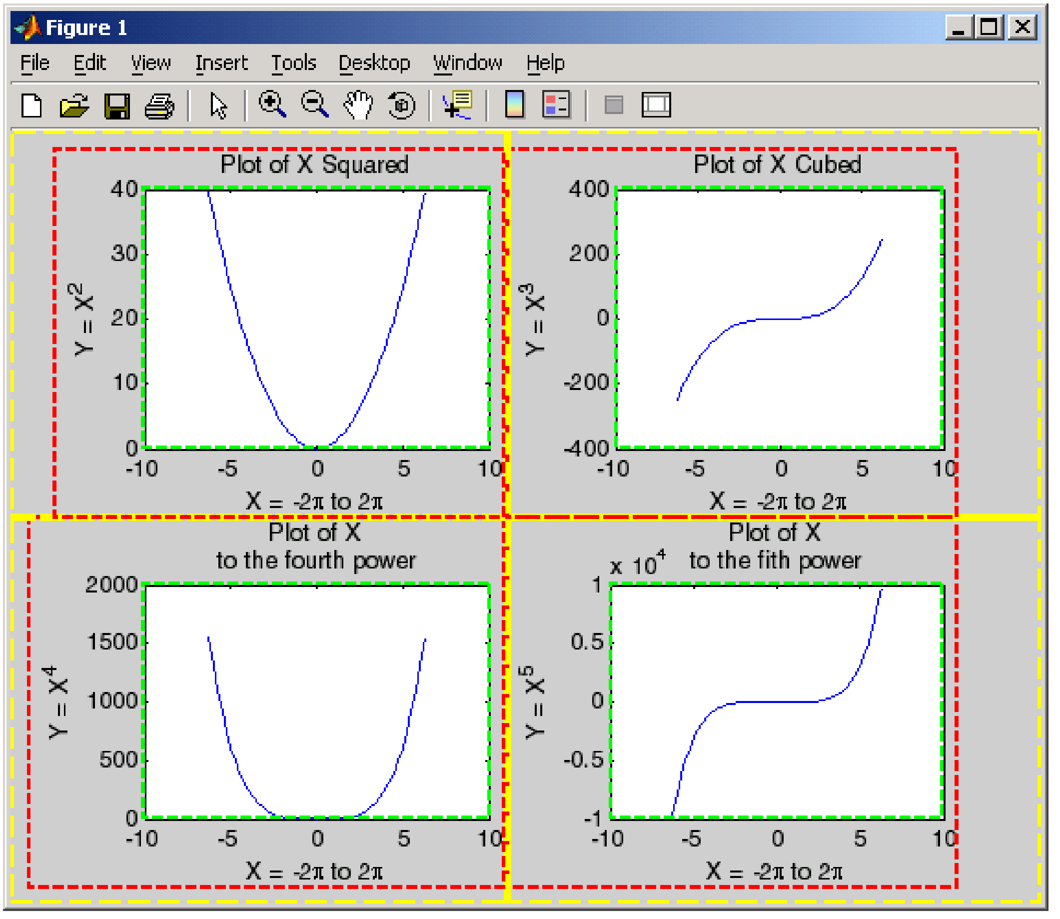
The following picture shows how these properties apply to 3-D graphs.



Axes Resizing in Subplots

Using the `OuterPosition` property as the `ActivePositionProperty` is an effective way to prevent titles and labels from being overwritten when there are multiple axes in a figure.

The following picture illustrates how MATLAB resizes the axes to accommodate the multiline titles on the lower two axes.



The default 3-D view is azimuth = -37.5°, elevation = 30°.

Multiple Axes per Figure

In this section...

Introduction

The `subplot` function creates multiple axes in one figure by computing values for `Position` that produce the specified number of axes.

The `subplot` function is useful for laying out a number of graphs equally spaced in the figure. However, overlapping axes can create some other useful effects. The following sections provide examples.

Placing Text Outside the Axes

The MATLAB software always displays text objects within an axes. If you want to create a graph and provide a description of the information alongside the graph, you must create another axes to position the text. If you create an axes that is the same size as the figure and then create a smaller axes to draw the graph, you can then display text anywhere independently of the graph.

For example, define two axes.

```
h = axes('Position',[0 0 1 1],'Visible','off');
axes('Position',[.25 .1 .7 .8])
```

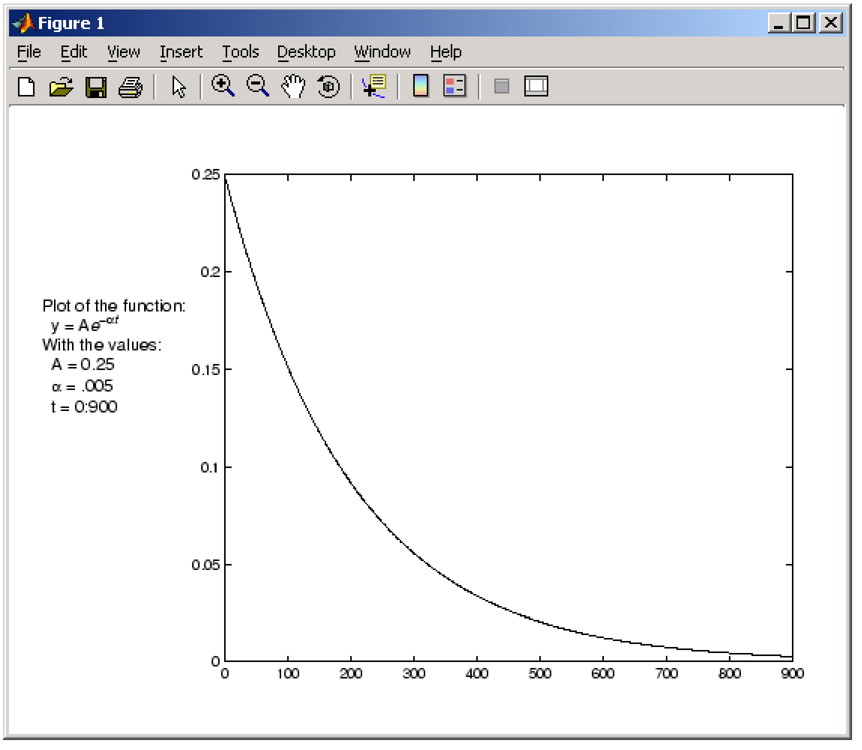
Because the axes units are normalized to the figure, specifying the `Position` as `[0 0 1 1]` creates an axes that encompasses the entire window.

Now plot some data in the current axes. The last axes created is the current axes, so MATLAB directs graphics output there.

```
t = 0:900;
plot(t,0.25*exp(-0.005*t))
```

Define the text and display it in the full-window axes.

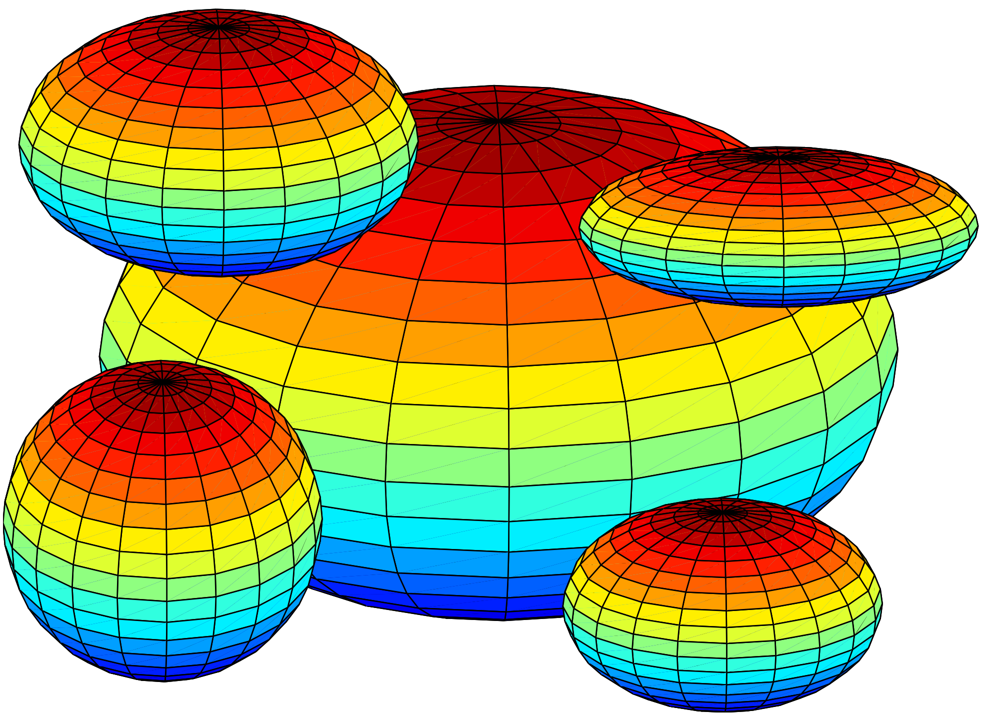
```
str(1) = {'Plot of the function:'};
str(2) = {' y = A{\ite}^{-\alpha{\itt}}'};
str(3) = {'With the values:'};
str(3) = {' A = 0.25'};
str(4) = {' \alpha = .005'};
str(5) = {' t = 0:900'};
set(gcf,'CurrentAxes',h)
text(.025,.6,str,'FontSize',12)
```



Multiple Axes for Different Scaling

You can create multiple axes to display graphics objects with different scaling without changing the data that defines these objects (which would be required to display them in a single axes).

```
h(1) = axes('Position',[0 0 1 1]);
sphere
h(2) = axes('Position',[0 0 .4 .6]);
sphere
h(3) = axes('Position',[0 .5 .5 .5]);
sphere
h(4) = axes('Position',[.5 0 .4 .4]);
sphere
h(5) = axes('Position',[.5 .5 .5 .3]);
sphere
set(h,'Visible','off')
```



Each sphere is defined by the same data. However, because the parent axes occupy regions of different size and location, the spheres appear to be different sizes and shapes.

Individual Axis Control

In this section...

Properties Controlling Axis Limits

The MATLAB software automatically determines axis limits, tick mark placement, and tick mark labels whenever you create a graph. However, you can specify these values manually by setting the appropriate property.

When you specify a value for a property controlled by a mode (e.g., the `XLim` property has an associated `XLimMode` property), MATLAB sets the mode to manual, enabling you to override automatic specification. Because the default values for these mode properties are automatic, calling high-level functions such as `plot` or `surf` resets these modes to `auto`.

This section discusses the following properties.

Property	Purpose
`XLim,YLim,ZLim`	Sets the axis range
`XLimMode,` `YLimMode,` `ZLimMode`	Specifies whether axis limits are determined automatically by MATLAB or specified manually by the user
`XTick,` `YTick,` `ZTick`	Sets the location of the tick marks along the axis

Property	Purpose
XTickMode, YTickMode, ZTickMode	Specifies whether tick mark locations are determined automatically by MATLAB or specified manually by the user
XTickLabel, YTickLabel, ZTickLabel	Specifies the labels for the axis tick marks
XTickLabelMode YTickLabelMode ZTickLabelMode	Specifies whether tick mark labels are determined automatically by MATLAB or specified manually by the user
XDir,YDir,ZDir	Sets the direction of increasing axis values

Setting Axis Limits

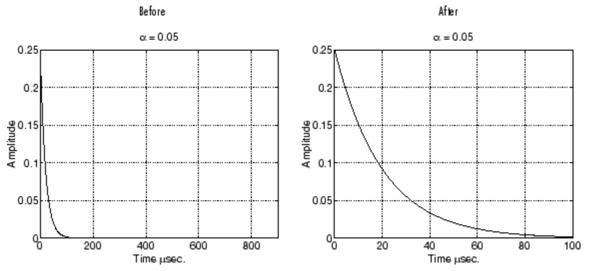
MATLAB determines the limits automatically for each axis based on the range of the data. You can override the selected limits by specifying the XLim, YLim, or ZLim property. For example, consider a plot of the function $Ae^{-\alpha t}$ evaluated with $A = 0.25$, $\alpha = 0.05$, and $t = 0$ to 900.

```
t = 0:900;
plot(t,0.25*exp(-0.05*t))
```

The plot on the left shows the results. MATLAB selects axis limits that encompass the range of data in both x and y. However, because the plot contains little information beyond $t = 100$, changing the x-axis limits improves the usefulness of the plot. If the handle of the axes is axes_handle, then the following statement,

```
set(axes_handle,'XLim',[0 100])
```

creates the plot on the right.



You can use the `axis` command to set limits on the current axes only.

Semiautomatic Limits

You can specify either the minimum or maximum value for an axis limit and allow the other limit to autorange. Do this by setting an explicit value for the manual limit and `Inf` for the automatic limit. For example, the statement

```
set(axes_handle,'XLim',[0 Inf])
```

sets the `XLimMode` property to `auto` and allows MATLAB to determine the maximum value for `XLim`. Similarly, the statement

```
set(axes_handle,'XLim',[-Inf 800])
```

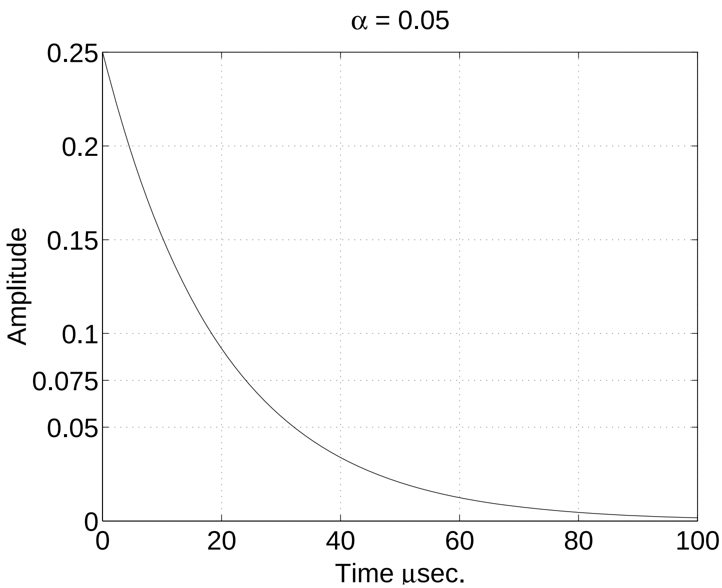
sets the `XLimMode` property to `auto` and allows MATLAB to determine the minimum value for `XLim`.

Setting Tick Mark Locations

MATLAB selects the tick mark location based on the data range to produce equally spaced ticks (for linear graphs). You can specify alternative locations for the tick marks by setting the `XTick`, `YTick`, and `ZTick` properties.

For example, if the value 0.075 is of interest for the amplitude of the function $Ae^{-\alpha t}$, specify tick marks to include that value.

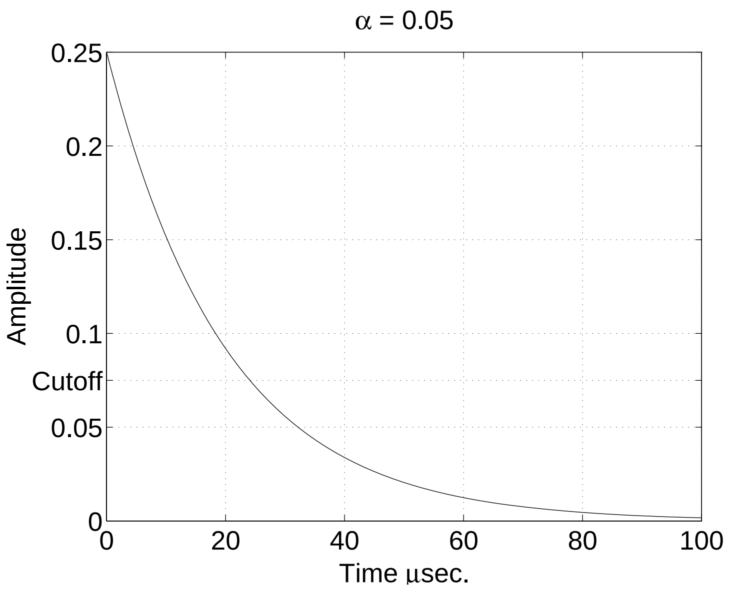
```
set(gca,'YTick',[0 0.05 0.075 0.1 0.15 0.2 0.25])
```



You can change tick labeling from numbers to strings using the `XTickLabel`, `YTickLabel`, and `ZTickLabel` properties.

For example, to label the *y*-axis value of 0.075 with the string `Cutoff`, you can specify all *y*-axis labels as a string, separating each label with the "`|`" character.

```
set(gca,'YTickLabel','0|0.05|Cutoff|0.1|0.15|0.2|0.25')
```



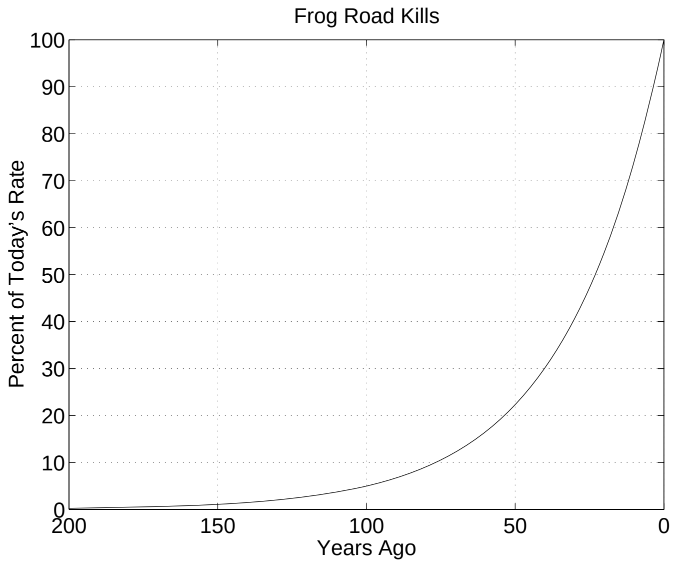
Changing Axis Direction

The `XDir`, `YDir`, and `ZDir` properties control the direction of increasing values on the respective axis. In the default 2-D view, the *x*-axis values increase from left to right and the *y*-axis values increase from bottom to top. The *z*-axis points out of the screen.

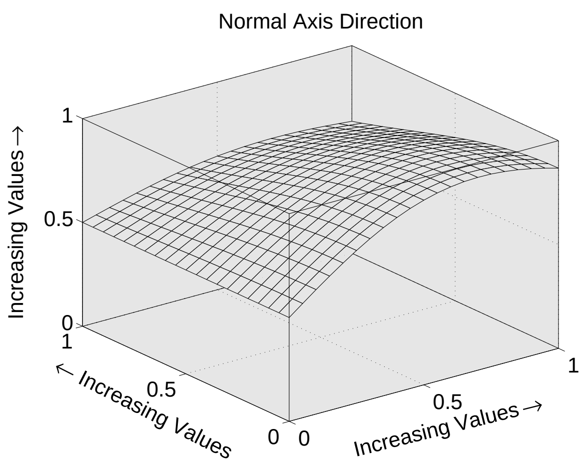
You can change the direction of increasing values by setting the associated property to `reverse`. For example, setting `XDir` to reverse,

```
set(gca,'XDir','reverse')
```

produces a plot whose *x*-axis decreases from left to right.



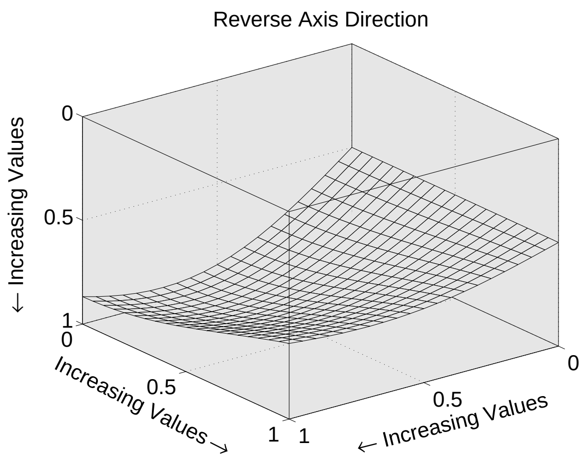
In the 3-D view, the *y*-axis increases from front to back and the *z*-axis increases from bottom to top.



Setting the *x*-, *y*-, and *z*-directions to reverse,

```
set(gca,'XDir','rev','YDir','rev','ZDir','rev')
```

yields



Using Multiple X- and Y-Axes

In this section...

Introduction

The `XAxisLocation` and `YAxisLocation` properties specify on which side of the graph to place the *x*- and *y*-axes. You can create graphs with two different *x*-axes and *y*-axes by superimposing two axes objects and using `XAxisLocation` and `YAxisLocation` to position each axis on a different side of the graph. This technique is useful to plot different sets of data with different scaling in the same graph.

Example — Double Axis Graphs

This example creates a graph to display two separate sets of data using the bottom and left sides as the *x*- and *y*-axis for one, and the top and right sides as the *x*- and *y*-axis for the other.

Suppose you have two sets of data having different *x*- and *y*-ranges:

```
x1 = [0:.1:40];
y1 = 4.*cos(x1)./(x1+2);
x2 = [1:.2:20];
y2 = x2.^2./x2.^3;
```

Using low-level `line` and `axes` routines allows you to superimpose objects easily. Plot the first data, making the color of the line and the corresponding *x*- and *y*-axis the same to more easily associate them.

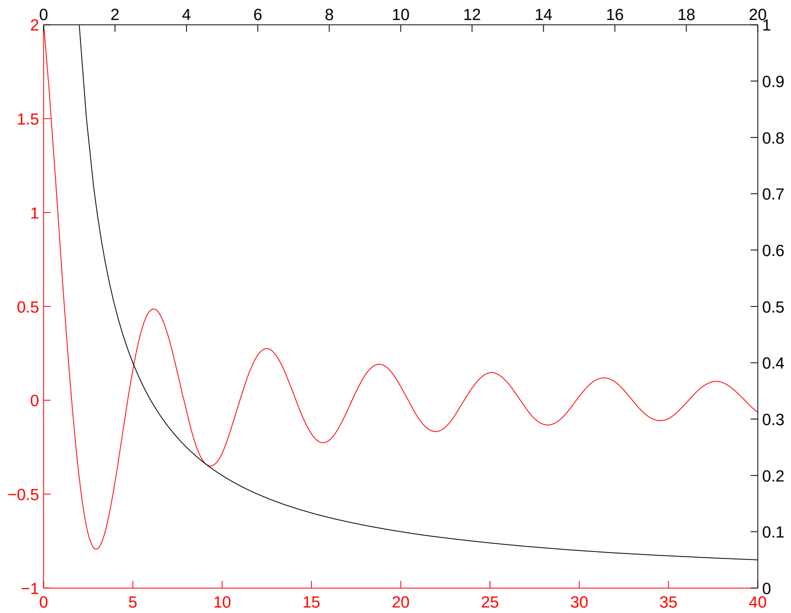
```
hl1 = line(x1,y1,'Color','r');
ax1 = gca;
set(ax1,'XColor','r','YColor','r')
```

Next, create another axes at the same location as the first, placing the *x*-axis on top and the *y*-axis on the right. Set the axes `Color` to `none` to allow the first axes to be visible and color code the *x*- and *y*-axis to match the data.

```
ax2 = axes('Position',get(ax1,'Position'),...
           'XAxisLocation','top',...
           'YAxisLocation','right',...
           'Color','none',...
           'XColor','k','YColor','k');
```

Draw the second set of data in the same color as the *x*- and *y*-axis.

```
hl2 = line(x2,y2,'Color','k','Parent',ax2);
```



Creating Coincident Grids

Since the two axes are completely independent, the MATLAB software determines tick mark locations according to the data plotted in each. It is unlikely the gridlines will coincide. This produces a somewhat confusing looking graph, even though the two grids are drawn in different colors. However, if you manually specify tick mark locations, you can make the grids coincide.

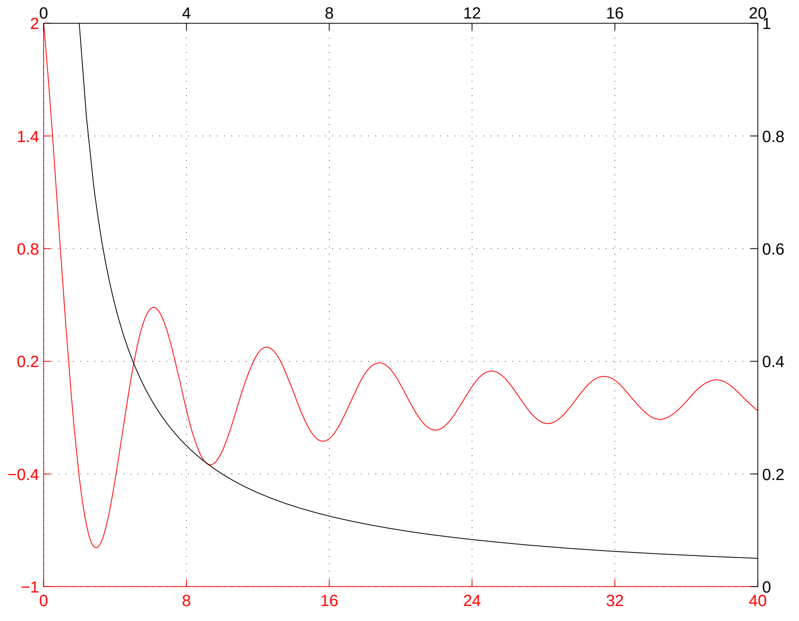
The key is to specify the same number of tick marks along corresponding axis lines (it is also necessary for both axes to be the same size). The following graph of the same data uses six tick marks per axis, equally spaced within the original limits. To calculate the tick mark locations, obtain the limits of each axis and calculate an increment.

```
xlimits = get(ax1,'XLim');
ylimits = get(ax1,'YLim');
xinc = (xlimits(2)-xlimits(1))/5;
yinc = (ylimits(2)-ylimits(1))/5;
```

Now set the tick mark locations.

```
set(ax1,'XTick',[xlimits(1):xinc:xlimits(2)],...
        'YTick',[ylimits(1):yinc:ylimits(2)])
```

Perform this calculation and set the axis limits for both axes. The resulting graph is visually simpler, even though the *y*-axis on the left has rather odd tick mark values.



If your data shares an *x*-axis, you can also use the `plotyy` function.

Automatic-Mode Properties

While object creation routines that create axes children do not explicitly change axes properties, some axes properties are under automatic control when their associated mode property is set to auto (which is the default). The following table lists the automatic-mode properties.

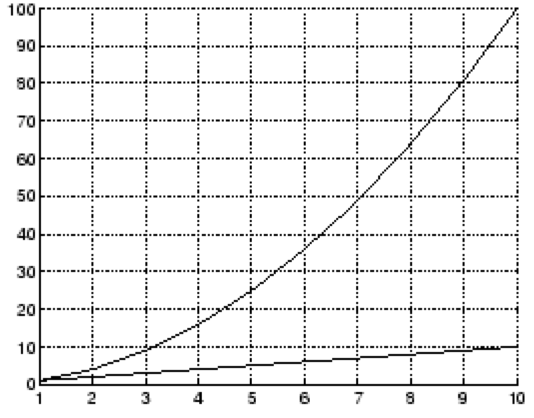
Mode Property	What It Controls
CameraPositionMode	Positioning of the viewpoint
CameraTargetMode	Positioning of the camera target in the axes
CameraUpVectorMode	The direction of "up" in 2-D and 3-D views
CameraViewAngleMode	The size of the projected scene and stretch-to-fit behavior
CLimMode	Mapping of data values to colors
DataAspectRatioMode	Relative scaling of data units along x-, y-, and z-axes and stretch-to-fit behavior
PlotBoxAspectRatioMode	Relative scaling of plot box along x-, y-, and z-axes and stretch-to-fit behavior
TickDirMode	Direction of axis tick marks (in for 2-D, out for 3-D)
XLimMode YLimMode ZLimMode	Limits of the respective x, y, and z axes
XTickMode YTickMode ZTickMode	Tick mark spacing along the respective x-, y-, and z-axes
XTickLabelMode ZTickLabelMode YTickLabelMode	Tick mark labels along the respective x-, y-, and z-axes

For example, if all property values are set to their defaults and you enter these statements,

```
line(1:10,1:10)
```

```
line(1:10,[1:10].^2)
```

the second `line` statement causes the `YLim` property to change from `[0 10]` to `[0 100]`.



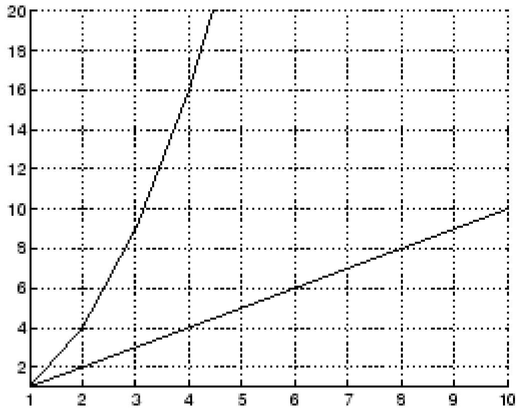
This is because `YLimMode` is `auto`, which always causes the MATLAB software to recompute the axis limits.

If you set the value controlled by an automatic-mode property, MATLAB sets the mode to `manual` and does not automatically recompute the value.

For example, in the statements

```
line(1:10,1:10)
set(gca,'XLim',[1 10],'YLim',[1 20])
line(1:10,[1:10].^2)
```

the `set` statement sets the *x*- and *y*-axis limits *and* changes the `XLimMode` and `YLimMode` properties to `manual`. The second `line` statement now draws a line that is clipped to the axis limits `[1 12]` instead of causing the axes to recompute its limits.


20
18
16
14
12
10
8
6
4
2
1
2
3
4
5
6
7
8
9
10

Colors Controlled by Axes

In this section...

Introduction

Axes properties specify the color of the axis lines, tick marks, labels, and the background. Properties also control the colors of the lines drawn by plotting routines and how image, patch, and surface objects obtain colors from the figure colormap.

The axes properties discussed in this section are listed in the following table.

Property	Characteristic it Controls
`Color`	Axes background color
`XColor`, `YColor`, `ZColor`	Color of the axis lines, tick marks, gridlines, and labels
`Title`	Title text object handles
`XLabel`, `YLabel`, `Zlabel`	Axis label text object handles
`CLim`	Controls mapping of graphic object `CData` to the figure colormap
`CLimMode`	Automatic or manual control of `CLim` property
`ColorOrder`	Line color autocycle order
`LineStyleOrder`	Line styles autocycle order (not a color, but related to `ColorOrder`)

Specifying Axes Colors

The default axes background color is set up by the `colordef` command, which is called in your startup file. However, you can easily define your own color scheme.

Changing the Color Scheme

Suppose you want an axes to use a “black-on-white” color scheme. First, change the background to white and the axis lines, grid, tick marks, and tick mark labels to black.

```
set(gca,'Color','w',...
        'XColor','k',...
        'YColor','k',...
        'ZColor','k')
```

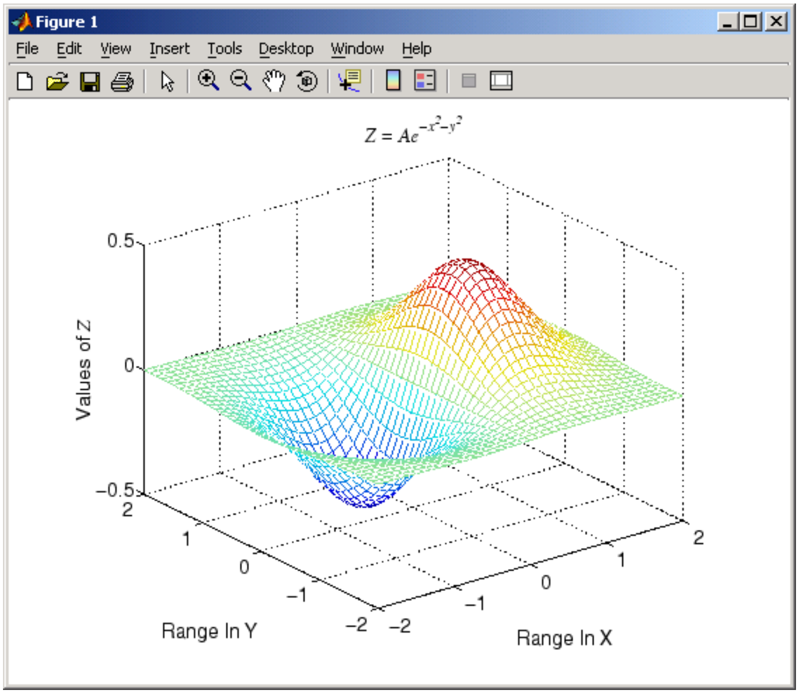
Next, change the color of the text objects used for the title and axis labels.

```
set(get(gca,'Title'),'Color','k')
set(get(gca,'XLabel'),'Color','k')
set(get(gca,'YLabel'),'Color','k')
set(get(gca,'ZLabel'),'Color','k')
```

Changing the figure background color to white completes the new color scheme.

```
set(gcf,'Color','w')
```

When you are done, a figure containing a mesh plot looks like the following figure.



You can define default values for the appropriate properties and put these definitions in your `startup.m` file. Titles and axis labels are text objects, so you must set a default color for all text objects, which is a good idea anyway because the default text color of white is not visible on the white background. Lines created with the low-level `line` function (but not the plotting routines) also have a default color of white, so you should change the default line color as well.

To set default values on the root level, use

```
set(0,'DefaultFigureColor','w'
       'DefaultAxesColor','w',...
       'DefaultAxesXColor','k',...
       'DefaultAxesYColor','k',...
       'DefaultAxesZColor','k',...
       'DefaultTextColor','k',...
       'DefaultLineColor','k')
```

The MATLAB software colors other axes children (i.e., image, patch, and surface objects) according to the values of their `CData` properties and the figure colormap.

Axes Color Limits — the CLim Property

In this section...

Introduction

Many 3-D plotting functions produce graphs that use color as another data dimension. For example, surface plots map surface height to color. The color limits control the limits of the color dimension in a way analogous to setting axis limits.

The axes `CLim` property controls the mapping of image, patch, and surface `CData` to the figure colormap. `CLim` is a two-element vector `[cmin cmax]` specifying the `CData` value to map to the first color in the colormap (`cmin`) and the `CData` value to map to the last color in the colormap (`cmax`). Data values in between are linearly transformed from the second to the penultimate color, using the expression

```
colormap_index = fix((CData-cmin)/(cmax-cmin)*cm_length)+1
```

`cm_length` is the length of the colormap. When the axes `CLimMode` property is `auto`, the MATLAB software sets `CLim` to the range of the `CData` of all graphics objects within the axes. However, you can set `CLim` to span any range of values. This enables individual axes within a single figure to use different portions of the figure's colormap. You can create colormaps with different regions, each used by a different axes.

See the `caxis` command for more information on color limits.

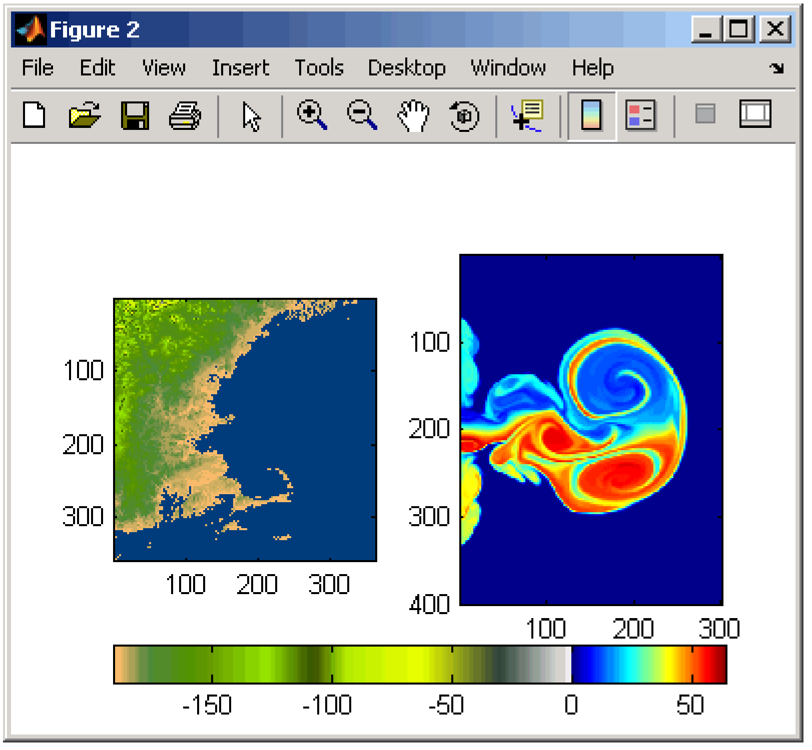
See "Introduction" on page 10-36 for an example that calculates color limits.

Simulating Multiple Colormaps in a Figure

Suppose you want to display two different images in the same figure. Images typically have their own colormaps, but you can specify only one colormap per figure. The solution is to concatenate the two colormaps and then setting the `CLim` property of each axes so that the two images map into different portions of the colormap.

Note **Colormap Size Limit.** On Windows platforms, the maximum length of the figure colormap is 256 colors (i.e., 256–by-3).

This example displays two images in one figure and maps the data in each image to the appropriate sections of the colormap, which has been created by concatenating the two colormaps together. The `colorbar` below the two images shows the entire colormap.



Complete Example Code

If you are using the MATLAB Help browser, you can:

- Run example
- Open the M-file in the editor

Calculating Color Limits

The key to this example is calculating values for `CLim` that cause each surface to use the section of the colormap containing the appropriate colors.

To calculate the new values for `CLim`, you need to know

- The total length of the colormap (`CmLength`)

- The beginning colormap slot to use for each axes (`BeginSlot`)
- The ending colormap slot to use for each axes (`EndSlot`)
- The minimum and maximum `CData` values of the graphic objects contained in the axes. That is, the values of the axes `CLim` property determined by MATLAB when `CLimMode` is `auto` (`CDmin` and `CDmax`).

First, define subplot regions and plot the surfaces.

```
im1 = load('cape.mat');
im2 = load('flujet.mat');
ax1 = subplot(1,2,1);
imagesc(im1.X)
axis(ax1,'image')
ax2 = subplot(1,2,2);
imagesc(im2.X)
axis(ax2,'image')
```

Concatenate two colormaps and install the new colormap.

```
colormap([im1.map;im2.map])
```

Obtain the data you need to calculate new values for `CLim`.

```
CmLength   = length(colormap);   % Colormap length
BeginSlot1 = 1;                  % Beginning slot
EndSlot1   = length(im1.map);    % Ending slot
BeginSlot2 = EndSlot1 + 1;
EndSlot2   = CmLength;
CLim1      = get(ax1,'CLim');  % CLim values for each axis
CLim2      = get(ax2,'CLim');
```

Defining a Function to Calculate CLim Values

Computing new values for `CLim` involves determining the portion of the colormap you want each axes to use relative to the total colormap size and scaling its `Clim` range accordingly. You can define a MATLAB function to do this.

```
function CLim = newclim(BeginSlot,EndSlot,CDmin,CDmax,CmLength)
   %     Convert slot number and range
```

```
    %      to percent of colormap
    PBeginSlot     = (BeginSlot - 1) / (CmLength - 1);
    PEndSlot       = (EndSlot - 1) / (CmLength - 1);
    PCmRange       = PEndSlot - PBeginSlot;
    %      Determine range and min and max
    %      of new CLim values
    DataRange      = CDmax - CDmin;
    ClimRange      = DataRange / PCmRange;
    NewCmin        = CDmin - (PBeginSlot * ClimRange);
    NewCmax        = CDmax + (1 - PEndSlot) * ClimRange;
    CLim           = [NewCmin,NewCmax];
end
```

The input arguments are identified in the bulleted list above. The function first computes the percentage of the total colormap you want to use for a particular axes (`PCmRange`) and then computes the `CLim` range required to use that portion of the colormap given the `CData` range in the axes. Finally, it determines the minimum and maximum values required for the calculated `CLim` range and returns these values. These values are the color limits for the given axes.

Using the Function

Use the `newclim` function to set the `CLim` values of each axes. The statement

```
set(ax1,'CLim',newclim(BeginSlot1,EndSlot1,CLim1(1),...
  CLim1(2),CmLength))
```

sets the `CLim` values for the first axes so the surface uses color slots 65 to 120. The lit surface uses the lower 64 slots. You need to reset its `CLim` values as well.

```
set(ax2,'CLim',newclim(BeginSlot2,EndSlot2,CLim2(1),...
  CLim2(2),CmLength))
```

How the Function Works

MATLAB enables you to specify any values for the axes `CLim` property, even if these values do not correspond to the `CData` of the graphics objects displayed in the axes. The minimum `CLim` value is always mapped to the first color in the colormap and the maximum `CLim` value is always mapped to the last

color in the colormap, whether or not there are really any `CData` values corresponding to these colors. Therefore, if you specify values for `CLim` that extend beyond the object's actual `CData` minimum or maximum, MATLAB colors the object with only a subset of the colormap.

The `newclim` function computes values for `CLim` that map the graphics object's actual `CData` values to the beginning and ending colormap slots that you specify. It does this by defining a "virtual" graphics object having the computed `CLim` values.

Defining the Color of Lines for Plotting

In this section...

Introduction

The axes `ColorOrder` property determines the color of the individual lines drawn by the `plot` and `plot3` functions. For multiline graphs, these functions cycle through the colors defined by `ColorOrder`, repeating the cycle when they reach the end of the list.

The `colordef` command defines various color order schemes for different background colors. `colordef` is typically called in the `matlabrc` file, which is executed during the MATLAB software startup.

Defining Your Own ColorOrder

You can redefine `ColorOrder` to be any *m*-by-3 matrix of RGB values, where *m* is the number of colors. However, high-level functions like `plot` and `plot3` reset most axes properties (including `ColorOrder`) to the defaults each time you call them. To use your own `ColorOrder` definition you must do one of the following three things:

- Define a default `ColorOrder` on the figure or root level
- Change the axes `NextPlot` property to `add` or `replacechildren`
- Use the informal form of the `line` function, which obeys the `ColorOrder` but does not clear the axes or reset properties

Changing the Default ColorOrder

You can define a new `ColorOrder` that MATLAB uses within a particular figure, for all axes within any figures created during the MATLAB session, or as a user-defined default that MATLAB always uses.

To change the `ColorOrder` for all plots in the current figure, set a default in that figure. For example, to set `ColorOrder` to the colors red, green, and blue, use the statement

```
set(gcf,'DefaultAxesColorOrder',[1 0 0;0 1 0;0 0 1])
```

To define a new `ColorOrder` that MATLAB uses for all plotting during your entire MATLAB session, set a default on the root level so axes created in any figure use your defaults.

```
set(0,'DefaultAxesColorOrder',[1 0 0;0 1 0;0 0 1])
```

To define a new `ColorOrder` that MATLAB always uses, place the previous statement in your `startup.m` file.

Setting the NextPlot Property

The axes `NextPlot` property determines how high-level graphics functions draw into an existing axes. You can use this property to prevent `plot` and `plot3` from resetting the `ColorOrder` property each time you call them, but still clear the axes of any existing plots.

By default, `NextPlot` is set to `replace`, which is equivalent to a `cla reset` command (i.e., delete all axes children and reset all properties, except `Position`, to their defaults). If you set `NextPlot` to `replacechildren`,

```
set(gca,'NextPlot','replacechildren')
```

MATLAB deletes the axes children, but does not reset axes properties. This is equivalent to a `cla` command without the `reset`.

After setting `NextPlot` to `replacechildren`, you can redefine the `ColorOrder` property and call `plot` and `plot3` without affecting the `ColorOrder`.

Setting `NextPlot` to `add` is the equivalent of issuing the `hold on` command. This setting prevents MATLAB from resetting the `ColorOrder` property, but it does not clear the axes children with each call to a plotting function.

Using the line Function

The behavior of the `line` function depends on its calling syntax. When you use the informal form (which does not include any explicit property definitions),

```
line(x,y,z)
```

`line` obeys the `ColorOrder` property, but does not clear the axes with each invocation or change the view to 3-D (as `plot3` does). However, `line` can be useful for creating your own plotting functions where you do not want the automatic behavior of `plot` or `plot3`, but you do want multiline graphs to use a particular `ColorOrder`.

Line Styles Used for Plotting — LineStyleOrder

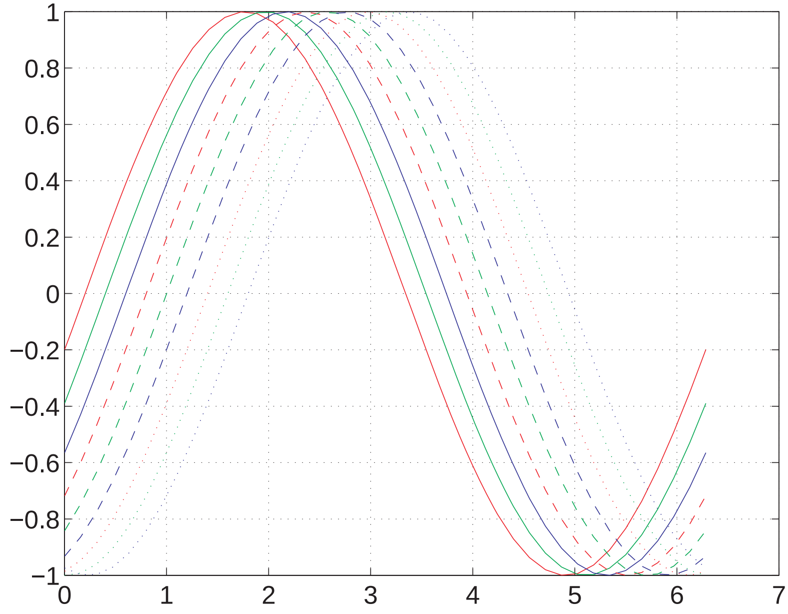
The axes `LineStyleOrder` property is analogous to the `ColorOrder` property. It specifies the line styles to use for multiline plots created with the `plot` and `plot3` functions. MATLAB increments the line style only after using all of the colors in the `ColorOrder` property. It then uses all the colors again with the second line style, and so on.

For example, define a default `ColoOrder` of red, green, and blue and a default `LineStyleOrder` of solid, dashed, and dotted lines.

```
set(0,'DefaultAxesColorOrder',[1 0 0;0 1 0;0 0 1],...
      'DefaultAxesLineStyleOrder','-|--|:')
```

Then plot some multiline data.

```
t = 0:pi/20:2*pi;
a = ones(length(t),9);
for i = 1:9
    a(:,i) = sin(t-i/5)';
end
plot(t,a)
```



MATLAB cycles through all colors for each line style.

Index

A

B

F

G

H

I

J

L

M

N

O

P

Q

R